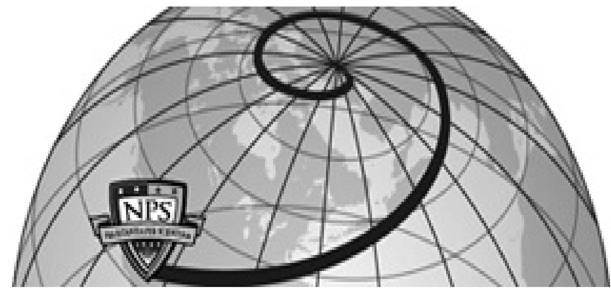
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Reports and Technical Reports | Faculty and Researchers' Publications

2020-12

Transonic Axial Compressor Design Tool

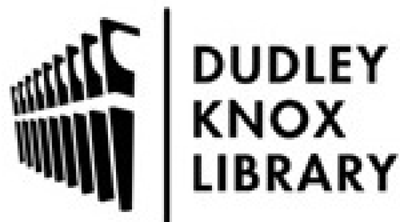
Gannon, Anthony J.; Hobson, Garth V.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/66349

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. As such, it is in the public domain, and under the provisions of Title 17, United States Code, Section 105, it may not be copyrighted.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NPS-MAE-20-001

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

TRANSONIC AXIAL COMPRESSOR DESIGN TOOL SOFTWARE

by

Anthony J. Gannon and Garth V Hobson

December 2020

Approved for public release; distribution is unlimited

Prepared for: U.S. Army Research Laboratory, Mechanical Sciences Division, Research Triangle Park, NC 27709-2211

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** ***(From-To)***
16 Dec 2020	Technical Report	12/31/2011 – 09/30/2015

4. TITLE AND SUBTITLE
TRANSONIC AXIAL COMPRESSOR DESIGN TOOL SOFTWARE

5a. CONTRACT NUMBER
MIPR2FDAVXR035

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Anthony J Gannon and Garth V Hobson

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-MAE-20-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory, Mechanical Sciences Division, Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A new design procedure using commercially-off-the-shelf (COTS) software, MATLAB, SolidWorks and ANSYS-WorkBench (CFX and Mechanical) for the geometry generation and analysis of axial compressors.

15. SUBJECT TERMS
Transonic, Axial Compressor, Stage, Design

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			
Unclassified	Unclassified	Unclassified	UU	143	Garth V Hobson
					19b. TELEPHONE NUMBER ***(include area code)*** (831) 656-2888

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ann E. Rondeau
President

Robert F. Dell
Acting Provost

The report entitled **TRANSONIC AXIAL COMPRESSOR DESIGN TOOL SOFTWARE** was prepared for U.S. Army Research Laboratory, Mechanical Sciences Division, Research Triangle Park, NC 27709-2211 and funded by U.S. Army Research Laboratory, Mechanical Sciences Division, Research Triangle Park, NC 27709-2211.

Further distribution of all or part of this report is authorized.

This report was prepared by:

____
Anthony J Gannon
Associate Professor

____
Garth V Hobson
Professor

Reviewed by:

____
Garth V Hobson, Chairman
Department of Mechanical and
Aerospace Engineering

Released by:

____
Jeffrey D. Paduan
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A new design procedure using commercially-off-the-shelf (COTS) software, MATLAB, SolidWorks and ANSYS-WorkBench (CFX and Mechanical) for the geometry generation and analysis of axial compressors.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

TPL Design Tool Software Documentation	**1**
Installation Instruction	**2**
Operation Overview	**5**
Function Operation and Use	**9**
Detailed File Guide	**14**
“Present Working Directory” File References	**17**
TPL Design Tool Software Flowchart	**18**
MainSpeedline_Auto.m	**20**
HardCodeBlade.m	**24**
HardCodeParams.m	**27**
GeomGen.m	**28**
BladeGen.m	**33**
BladeSect.m	**36**
Polygeom.m	**41**
SangerMethod.m	**44**
Passage.m	**57**
WedgeGen.m	**61**
StreamGen.m	**66**
d_d.m	**70**
NEWS.m	**73**
extrap.m	**76**
SolidWorksGen.m	**77**
BladeHub_Wedge_CutOut.m	**90**
FluidAnalysis.m	**106**
UpdateProject1.py	**109**
DM_CAD_Refresh.js	**113**
UpdateProject2.py	**114**
ReadAnsysData.m	**118**
ArchiveRunData.m	**125**
ArchiveProject.py	**127**
WriteSpreadSheet.m	**128**

This page intentionally left blank

This page intentionally left blank

I. TPL DESIGN TOOL SOFTWARE DOCUMENTATION

Important Files

1. AirWedge.x_t
2. ArchiveProject.py
3. ArchiveRunData.m
4. BasicRotor.SLDPRT
5. BasicWedge.SLDPRT
6. BladeGen.m
7. BladeHub_Wedge_CutOut.m
8. BladeSect.m
9. d_d.m
10. DM_CAD_Refresh.js
11. extrap.m
12. FluidAnalysis.m
13. GeomGen.m
14. HardCodeBlade.m
15. HardCodeParams.m
16. Main_Single.m
17. Main_SpeedLine_Auto.m
18. Main_SpeedLine_ManualContinue.m
19. MinDist.m
20. NEWS.m
21. Passage.m
22. polygeom.m
23. ReadAnsysData.m
24. SangerMethod.m
25. SolidWorksGen.m
26. StreamGen.m
27. UpdateProject1.py
28. UpdateProject2.py
29. WedgeGen.m
30. WorkingProject.wbpj [Also include corresponding folder of the same name]
31. WriteSpreadsheet.m

Installation Instructions

****** IMPORTANT ******: *Before copying files to a new location, the* ***WorkingProject.wbpj*** *file must be 'archived' using the ANSYS Workbench 'Archive' function (File tab). Once the* ***WorkingProject.wbpz*** *file is copied to the desired folder, it must be 'unarchived' and saved as "****WorkingProject.wbpj****" using the ANSYS Workbench 'Restore Archive' function (File tab).*

Copy **AirWedge.xt** and all the .m (MATLAB) files to the same folder. The other files may be copied elsewhere, but for ease of setup and use they should be copied into the same folder as the .m files.

After copying, edit the following files:

- **SolidWorksGen.m**
  - Line 13: Change the file path to the current location of BasicWedge.SLDPRT or [pwd '\BasicWedge.SLDPRT']. (pwd = present working directory)
- **BladeHub_Wedge_CutOut.m**
  - Line 13: Change the file path to the current location of BasicRotor.SLDPRT or [pwd '\BasicRotor.SLDPRT']. (pwd = present working directory)
- **FluidAnalysis.m**
  - Line 23: Change "filePath" to the same location as WorkingProject.wbpj. Unlike the previous file paths, this should have no file name at the end, just a backslash.
  - Line 44: Change the file path after the -F flag to the current location of WorkingProject.wbpj.
  - Line 44: Change the file path after the -R flag to the current location of UpdateProject.py.
  - Line 46: Change the file path after the -F flag to the current location of WorkingProject.wbpj.
  - Line 46: Change the file path after the -R flag to the current location of UpdateProject.py.

- **UpdateProject1.py**

  Tip: Opening this file in the MATLAB program editor rather than Notepad will display the line numbers, making it easier to edit.

  - Line 4: Change the file path to the same location as WorkingProject.wbpj. This must match Line 23 of FluidAnalysis.m and end in "\\".
  - Line 12: Change the file path to the current location of DM_CAD_Refresh.js.

- **UpdateProject2.py**

  Tip: Opening this file in the MATLAB program editor rather than Notepad will display the line numbers, making it easier to edit.

  - Line 4: Change the file path to the same location as WorkingProject.wbpj. This must match Line 23 of FluidAnalysis.m and end in "\\".

- **ArchiveRunData.m**
  - Line 59: Change the file path to the same location as WorkingProject.wbpj.
  - Line 69: Change the file path after the -F flag to the current location of WorkingProject.wbpj.
  - Line 69: Change the file path after the -R flag to the current location of ArchiveProject.py.

- **ArchiveProject.py**

  Tip: Opening this file in the MATLAB program editor rather than Notepad will display the line numbers, making it easier to edit.

  - Line 6: Change the file path to the same location as WorkingProject.wbpj. This must point to the same location as Line 42 of ArchiveRunData.m.

- **WorkingProject.wbpj**
    - Open in ANSYS Workbench. Edit the Geometry, and change the file path under the details of "Import1" to the current location of AirWedge.x_t. Click "Generate."
        - **Note:** Do not change the file path by selecting "Replace Geometry" in Workbench. Doing so will cause problems by changing the locations of named selections and match controls later in the project.
    - Optional: Configure other settings like meshing relevance center and number of iterations per run.
    - Save the project with the new settings.

II. OPERATION OVERVIEW

Running Main_Single generates a rotor geometry and runs a single fluid analysis.
Main_SpeedLine_Auto automatically creates the geometry and generates a single speed line until pressure ratio or power drops, while Main_SpeedLine_ManualContinue prompts the user whether or not to continue after every second generated point.
The two main functions are GeomGen and FluidAnalysis. GeomGen generates and saves the following parasolid files for stress and fluid analyses:

- BladeHub_Full.x_t: Complete rotor
- BladeHub_Wedge_Cutout.x_t: Portion of rotor inside the air wedge
- AirWedge.x_t: Air wedge with no fillets
- AirWedge_Fillets.x_t: Air wedge with fillets
- AirWedge_Upstream.x_t: Half of the air wedge, upstream of the rotor trailing edge
- AirWedge_Downstream.x_t: Half of the air wedge, downstream of the rotor trailing edge

The geometries are also saved as SolidWorks part files (.SLDPRT) with matching names.
Warning: As a precaution, GeomGen first closes all open SolidWorks files without asking to save. Save and close all work before running GeomGen.
Caution: Do not click in the SolidWorks window while the geometry is built. Doing so may cause errors in the final model.

FluidAnalysis opens WorkingProject.wbpj, updates the project geometry with the current version of AirWedge.x_t, reruns the analysis with previously saved settings, and resaves the project. Data for the current run is appended onto previous run data. MATLAB remains occupied while a simulation runs and will not accept commands until ANSYS exits. Once control returns to MATLAB, FluidAnalysis reads and returns output parameters from SavedOutput.dat, which contains values generated by ANSYS.
GeomGen and FluidAnalysis can be run independent of each other as long as a geometry file is present for FluidAnalysis to analyze. As runs are performed, WorkingProject.wbpj gradually accumulates data. This can be cleared by right-clicking “Solution” in Workbench and selecting “Clear Generated Data” or by running ArchiveRunData, a separate function that archives and

saves files from simulation runs. See the below section on “Function Operation and Use” for more details.

Warning: Several .dat files are used to pass information between MATLAB and ANSYS. Do not delete them while the program is running. There is nothing to prevent them from being deleted between the time they are created and the time they are read, and deleting them while functions are running could cause errors. Deleting them when the functions are not running, however, should cause no problems.

GeomGen(Blade, Params)

Input

Blade:

III. FUNCTION OPERATION AND USE

Structure containing parameters that describe the rotor blade(s), requiring the following fields:

PassNo: Number of blade passages; number of times the calculated blade profiles are repeated around the hub

S: Number of sections used to generate the blade

P: Number of points that define a blade profile. One blade profile actually contains twice this number of points.

Heights: Specifies heights for property inputs

Entered as a vertical vector

Chord: Passage chord at "Heights" location

One column for each blade element

LE: Blade leading edge shift backwards/forwards at "Heights" as a fraction of axial chord

One column for each blade element

Edges: Blade leading and trailing edge at "Heights"

Entered as a three-dimensional array

Rows are in the form [Leading edge minor axis/Chord | Leading edge eccentricity | Trailing edge minor axis/Chord | Trailing edge eccentricity]

Each row matches a height in "Heights"

Third dimension matches blade element

Controls: Chord control point locations

Entered as a horizontal vector

Stagger: Stagger of blade elements

Entered as a three-dimensional array

Rows match "Heights"

Columns match chord control point locations

Third dimension matches blade element

Thickness: Blade element thickness

Entered as a three-dimensional array

Rows match heights

Columns match chord control point locations

Third dimension matches blade element

Offset: Fraction of passage to rotate each blade element

Fraction = 1 would rotate across the entire passage

Rows are in the form [radius | fraction]

Columns match heights

MasterXShift: Additional distance in inches to shift all blades in the x-direction.

Fillet: Fillet radius of the blade in inches

Center: Boolean specifying whether to center the primary blade on the hub origin (true) or align the leading edge with the origin (false) before MasterXShift is applied

Params:

Structure containing other constants and parameters, requiring the following fields:

Air.Cp: Coefficient of specific heat

Air.Gam: Specific heat ratio of air

Inlet.Po: Standard atmospheric pressure at inlet in Pa

Inlet.To: Standard atmospheric temperature in K

PR: Design pressure ratio

RPM: Design RPM

W: Specific Weightflow

rho1: Ambient Density

R: Gas Constant in kJ/(kg K)

eta: Rotor Efficiency

AR: Aspect Ratio

SA: Midspan Stagger Angle in degrees

AVR: Axial Velocity Ratio

TIR: Rotor Tip Inlet Radius in meters

TER: Rotor Tip Exit Radius in meters

HER: Rotor Hub Exit Radius in meters

OCD: Outer Casing Diameter in meters

OT_axial: Axial Outlet Hub Transition in meters

OT_radial: Radial Outlet Hub Transition in meters

UpstreamAxial: Upstream axial wedge length in number of casing diameters

DownstreamAxial: Downstream axial wedge length in number of casing diameters

The necessary fields for both structures can also be seen by typing "help GeomGen" into MATLAB. HardCodeBlade is a sample function that generates and returns a Blade for a basic Sanger rotor, while HardCodeParams generates and returns a sample Params object.

Prerequisites

N/A

Return Values

N/A

Operation

Generates points and sends commands to generate rotor geometry in SolidWorks.

FluidAnalysis(OutletPressure, AngularVelocity, PassNo)

Inputs

OutletPressure:

Outlet pressure in atmospheres to use in the CFX analysis

AngularVelocity:

Angular velocity in revolutions per minute to use in the CFX analysis

Note: This must be entered as a negative value, such as "-30000" for 30000 rpm.

PassNo:

Number of blade passages. Should be identical to Blade.PassNo used to build the geometry.

Prerequisites

AirWedge.x_t produced by GeomGen must exist for analysis.

Return Values

outputs:

Structure containing the following fields:

effTT: Total-to-total isentropic efficiency

pTotalOut: Total Pressure, Outlet

pTotalOutUnits: Units of pTotalOut

pTotalIn: Total Pressure, Inlet

pTotalInUnits: Units of pTotalIn

mFlowIn: Total mass flow at inlet for the overall rotor

mFlowInUnits: Units of mFlowIn

mFlowOut: Total mass flow at outlet for the overall rotor

mFlowOutUnits: Units of mFlowIn

power: Total power for all blades.

powerUnits: Units of power

Note: After FluidAnalysis runs, these same output values can also be found in the file SavedOutput.dat, which is overwritten for each run. However, this file contains the mass flow and power for only one blade passage, not the total mass flow/power. Similarly, ReadAnsysData returns these values for only the single passage.

Operation

Writes a text file containing outlet pressure and angular velocity, activates an ANSYS Workbench script which updates WorkingProject.wbpj with those settings, and returns the output data specified above.

> **Note:** While displaying the current solution run, CFX Solver will display the "Define Run" box. No action needs to be taken on the user's part, and this box can simply be closed. Inadvertently selecting "Start Run" will not adversely affect the run already in progress, although Workbench may complain of an error.

ArchiveRunData

Input

N/A, runs as a script rather than a function.

Prerequisites

As written, requires that all backed-up files be in the present working directory. This behavior can be changed by altering the source in the DOS Copy command.

Returns

N/A

Operation

Makes a new directory with the current date stamp and copies all .m, .js, .py, .x_t, and .SLDPRT files into the new directory. (Except for the ANSYS Workbench project, these should be all the files necessary to later rerun the program in its current state.) Triggers an ANSYS script to save WorkingProject.wbpj into an archive file in the new directory and, once that is complete, clear all existing run data and messages from WorkingProject.wbpj in preparation for future runs.

IV. DETAILED FILE GUIDE

AirWedge.x_t: Parasolid file of an air wedge without fillets. Overwritten by GeomGen during geometry generation. Included in list of important files because one must already exist to provide WorkingProject.wbpj with the file path.

ArchiveProject.py: ANSYS script that archives and clears WorkingProject.wbpj for future runs.

ArchiveRunData.m: Copies all project files and runs ArchiveProject.py.

BasicRotor.SLDPRT: Blank SolidWorks part file that serves as a template for the rotor.

BasicWedge.SLDPRT: Blank SolidWorks part file that serves as a template for the air wedge.

BladeGen.m: Calculates blade profiles for solid generation.

BladeHub_Wedge_Cutout.m: When run inside GeomGen.m, this generates the full rotor, the segment of the rotor inside the air wedge, and the air wedge with fillets.

BladeSect.m: Calculates individual blades.

d_d.m: Finds the first derivative of two variables.

DM_CAD_Refresh.js: JScript file run in DesignModeler to update the new part geometry in the project. Supplied by ANSYS technical support.

Main_Single.m: Sample program that runs GeomGen and Fluid Analysis.

extrap.m: Performs quadratic extrapolation to the bottom of a matrix.

FluidAnalysis.m: Starts ANSYS, opens the project file, and runs UpdateProject.py.

GeomGen.m: Runs all code to generate the rotor geometry in SolidWorks.

HardCodeBlade.m: Generates a sample “Blade” structure to be used by GeomGen.

HardCodeParams.m: Generates a sample “Params” structure to be used by GeomGen.

Main_SpeedLine_Auto.m: Sample program that generates a speed line. Automatically increases outlet pressure until the pressure ratio or power begin to drop or have a nonexistent value, indicating a probable stall. Initially uses an increment of 0.05 atm, then decreases to 0.01 atm when the outlet pressure reaches 0.35 atm.

Main_SpeedLine_ManualContinue.m: Operates similarly to Main_Speedline_Auto.m, but asks the user to continue after every other iteration rather than making an automatic decision.

NEWS.m: Finds north, south, east, and west points from a non-uniform grid.

Passage.m: Generates the blade passage.

polygeom.m: Generates properties of a planar polygon.

ReadAnsysData.m: Reads data from output file written by UpdateProject.py.

SangerMethod.m: Generates a hub profile with conical spinner. Incorporates Sanger design methodology with shock loss model. Also calculates the mass flow rate used in GeomGen.

SolidWorksGen.m: When run from GeomGen.m, generates the full air wedge without fillets as well as the upstream and downstream halves of the air wedge (also without fillets).

StreamGen.m: Calculates streamline radial positions.

UpdateProject1.py: ANSYS script to update the CFX project with the new geometry and generate a new mesh for initial run.

UpdateProject2.py: ANSYS script to update the CFX project for subsequent runs (not initial).

WedgeGen.m: Outputs the gas wedge geometry and ensures it contains the rotor blades.

WorkingProject.wbpj: ANSYS fluid analysis project file.

WorkingProject_files: Folder containing files used by WorkingProject.wbpj.

WriteSpreadsheet.m: Writes collected output data to a spreadsheet.

V. <u>**“PRESENT WORKING DIRECTORY” FILE REFERENCES**</u>

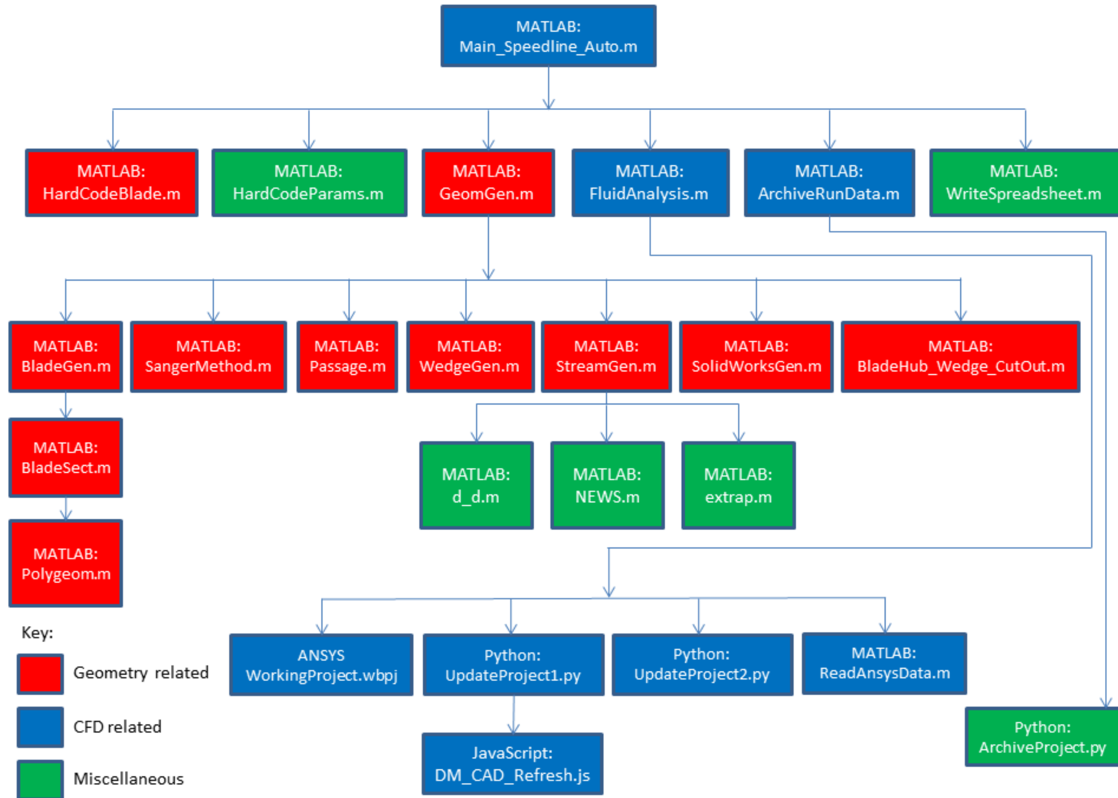
The following file lines use the Present Working Directory (“pwd”) to simplify file paths:

- **SolidWorksGen.m**
  - Line 13: BasicWedge.SLDPRT
  - Line 483: AirWedge.x_t
  - Line 484: AirWedge.SLDPRT
- **BladeHub_Wedge_CutOut.m**
  - Line 13: BasicRotor.SLDPRT
  - Line 296: BladeHub_Full.x_t
  - Line 297: BladeHub_Full.SLDPRT
  - Line 547: BladeHub_Wedge_CutOut.x_t
  - Line 548: BladeHub_Wedge_CutOut.SLDPRT
  - Line 577: AirWedge_Fillets.x_t
  - Line 578: AirWedge_Fillets.SLDPRT
- **ArchiveRunData.m**
  - Line 11: Final save location

  While the following lines do not directly refer to the present working directory, its use is implied in the source of the Copy commands.

  - Line 28: *.m
  - Line 30: *.py
  - Line 32: *.js
  - Line 34: *.x_t
  - Line 36: *.SLDPRT

VI. TPL DESIGN TOOL SOFTWARE FLOWCHART



This page intentionally left blank

VII. MAIN_SPEEDLINE_AUTO.M

```
% Generates points for a speed line for a rotor

clear all
close all

Blade = HardCodeBlade_Hobson_R13; % Loads sample blade properties
Params = HardCodeParams; % Loads sample parameters
GeomGen(Blade, Params); % Generates the blade geometry

pOutlet = 0.000             % Set the initial outlet pressure
pOutletIncrement = 0.200;   % At least to begin with, by how much do we
increment the outlet pressure?
pOutThreshold1 = 0.400;      % At what outlet pressure do we change the
speedline increment?
pOutThreshold2 = 0.500;      % At what outlet pressure do we change the
speedline increment?
%rpm = -30000;               % At what angular velocity are we running the
simulation?--Disabled to use Params.RPM instead

% Initialize arrays to store properties to null
effTT = [];
mFlowIn = [];
mFlowOut = [];
pTotalOut = [];
pTotalIn = [];
pRatio = [];
power = [];
omega = [];

% Generate blank graphs
% Pressure Ratio vs. Mass Flow
figure(9); close; figure(9);
title('Pressure Ratio vs. Mass Flow')
xlabel('Mass Flow', 'Interpreter', 'none')
ylabel('Pressure Ratio', 'Interpreter', 'none')
hold on;
pRatioInPlot = plot(mFlowIn, pRatio, 'b');
pRatioOutPlot = plot(mFlowOut, pRatio, 'g-.');

% Isentropic Efficiency vs. Mass Flow
figure(10); close; figure(10);
title('Total-to-Total Isentropic Efficiency vs. Mass Flow')
xlabel('Mass Flow', 'Interpreter', 'none')
ylabel('Total-to-Total Isentropic Efficiency', 'Interpreter', 'none')
hold on;
effTTInPlot = plot(mFlowIn, effTT, 'r');
effTTOutPlot = plot(mFlowIn, effTT, 'k-.');

% Power vs. Mass Flow
figure(11); close; figure(11);
title('Power (all blades) vs. Mass Flow')
```

```
xlabel('Mass Flow', 'Interpreter', 'none')
ylabel('Power (all blades)', 'Interpreter', 'none')
hold on;
powerInPlot = plot(mFlowIn, power, 'r');
powerOutPlot = plot(mFlowIn, power, 'b-.');

proceed = true;
ii = 1;

% Enclose in a try-catch block so that if an error occurs all existing data
% will be saved.
try
    while proceed
        if ii ~= 1       % Don't add an outlet pressure the first time
            pOutlet = [pOutlet (pOutlet(end) + pOutletIncrement)];
        end
        omega = [omega -Params.RPM];
        outputs = FluidAnalysis(pOutlet(ii), omega(ii), Blade.PassNo, ii); %
Analyzes the blade
        effTT = [effTT outputs.effTT];
        mFlowIn = [mFlowIn outputs.mFlowIn];
        mFlowOut = [mFlowOut outputs.mFlowOut];
        pTotalOut = [pTotalOut outputs.pTotalOut];
        pTotalIn = [pTotalIn outputs.pTotalIn];
        pRatio = [pRatio outputs.pTotalOut/outputs.pTotalIn];
        power = [power outputs.power];


        %-----------------------------------------------------------------------
------
        % Generate all graphs

        % Plot pressure ratio vs. inlet mass flow
        figure(9);
        xlabel(['Mass Flow ' outputs.mFlowInUnits], 'Interpreter', 'none')
        % If plot handle is empty (i.e., there is no plotted data), plot
        % data fresh.  Otherwise, add the new data on to the end
        if isempty(pRatioInPlot)
            pRatioInPlot = plot(mFlowIn, pRatio, 'b');
        else
            set(pRatioInPlot, 'XData', mFlowIn, 'YData', pRatio);
        end
        if isempty(pRatioOutPlot)
            pRatioOutPlot = plot(mFlowOut, pRatio, 'g-.');
        else
            set(pRatioOutPlot, 'XData', mFlowOut, 'YData', pRatio);
        end

        axis auto;
        v = axis;            % Save the current axes for altering
        v(3) = 1;            % Make it so that the y-axis starts at one
        v(4) = 1.1*v(4);     % Add ten percent to the y-axis
        axis(v)
```

```
        text(mFlowIn(ii), pRatio(ii), ['\bullet ' num2str(pOutlet(ii)) '
[atm]']);

        legend('Pressure ratio vs. Inlet mass flow','Pressure ratio vs.
Outlet mass flow', 'Location', 'SouthOutside')

        % Plot isentropic efficiency vs. inlet mass flow
        figure(10);
        xlabel(['Mass Flow ' outputs.mFlowInUnits], 'Interpreter', 'none')
        if isempty(effTTInPlot)
            effTTInPlot = plot(mFlowIn, effTT, 'r');
        else
            set(effTTInPlot, 'XData', mFlowIn, 'YData', effTT);
        end
        if isempty(effTTOutPlot)
            effTTOutPlot = plot(mFlowOut, effTT, 'k-.');
        else
            set(effTTOutPlot, 'XData', mFlowOut, 'YData', effTT);
        end

        axis auto;
        v = axis;             % Save the current axes for altering
        v(3) = 0;             % Make it so that the y-axis starts at zero
        v(4) = 1;             % Since this is efficiency, go up to 1
        axis(v)

        % Label the points with the outlet pressure
        text(mFlowIn(ii), effTT(ii), ['\bullet ' num2str(pOutlet(ii)) '
[atm]']);

        legend('Isentropic efficiency vs. Inlet mass flow','Isentropic
efficiency vs. Outlet mass flow', 'Location', 'SouthOutside')

        % Plot power vs. inlet mass flow
        figure(11);
        xlabel(['Mass Flow ' outputs.mFlowInUnits], 'Interpreter', 'none')
        ylabel(['Power (all blades) ' outputs.powerUnits], 'Interpreter',
'none')
        if isempty(powerInPlot)
            powerInPlot = plot(mFlowIn, power, 'r');
        else
            set(powerInPlot, 'XData', mFlowIn, 'YData', power);
        end
        if isempty(powerOutPlot)
            powerOutPlot = plot(mFlowOut, power, 'b-.');
        else
            set(powerOutPlot, 'XData', mFlowOut, 'YData', power);
        end

        axis auto;
        %v = axis;             % Save the current axes for altering
        %%v(3) = 0;            % Make it so that the y-axis starts at zero
        %v(3) = 0.9*v(3);      % Subtract ten percent from the bottom of the y-
axis
        %v(4) = 1.1*v(4);      % Add ten percent to the y-axis
```

```
        %axis(v)

        % Label the points with the outlet pressure
        text(mFlowIn(ii), power(ii), ['\bullet ' num2str(pOutlet(ii)) '
[atm]']);

        legend('Power (all blades) vs. Inlet mass flow','Power (all blades)
vs. Outlet mass flow', 'Location', 'SouthOutside')

        % Finish when pRatio is below its previous value or is not a valid
number
        % Only check when ii > 1 to prevent errors
        if ii > 1
            if (pRatio(ii) < pRatio(ii-1))|| isnan(pRatio(ii))
                proceed = false;
            end
        end


        if pOutlet(end) >= 0.200
            proceed = false;
        end

        if pOutlet(end) >= pOutThreshold1
            pOutletIncrement = 0.050;
        end

        if pOutlet(end) >= pOutThreshold2
            pOutletIncrement = 0.010;
        end

        ii = ii + 1;
    end

%-----------------------------------------------------------------------------
% Save data into an Excel spreadsheet, even if an error occurs
catch err
    disp('ERROR: Analysis Failed')
    disp(getReport(err))
    ArchiveRunData
    WriteSpreadsheet
    rethrow(err)
end
ArchiveRunData
WriteSpreadsheet
```

VIII. HARDCODE BLADE.M

```
% Function that generates a hard-coded "Blade"

function Blade = HardCodeBlade_Hobson_R13
Blade.PassNo    = 12;                          % Number of passages
Blade.S         = 9;                           % Number of blade sections
Blade.P         = 60;                         % Number of points defining the
blade profile

% Control vector dimensions must agree with 'Heights' and 'Chords'
Blade.Heights   = [2.8 4.5 5.7]';                        % Control Input heights

Blade.Chord(:,1)     = [3.5;  2.0;  5.0];             % Passage chord at
'Heights' location  [3.0;  2.0;  5.0];
Blade.Chord(:,2)     = [1.5;  1.5;  2.5];             % Passage chord at
'Heights' location

Blade.LE        = [0.0  0.2;...
                   0.0  0.2;...
                   0.0  0.2;];         % Blade leading edge shift as fraction
of axial chord, one column for each blade element

% Blade leading and trailing edge [Lead minor axis/Chord; eccen; Trail minor
axis/Chord; eccen;]
Blade.Edges(:,:,1) = [0.06  2 0.04  1;  % [0.04  2 0.04  1; Massively reduces
leading edge stress
                      0.03  2 0.02  1;  %  0.02  2 0.02  1;
                      0.01  2 0.02  1];

% Blade leading and trailing edge [Lead minor axis/Chord; eccen; Trail minor
axis/Chord; eccen;]
Blade.Edges(:,:,2) = [0.04   2 0.04  1;
                      0.02   2 0.02  1;
                      0.01   2 0.02  1];

% Chord control point locations
Blade.Controls    = [0 0.3 0.7 1];                  % Control Input chords
% 1 stagger matrix per element
Blade.Stagger(:,:,1) = [65 45 20 15;...   % 65 45 15 10;...
                        75 65 25 20;...   % 75 65 25 15;...
                        75 72 70 65];     % 1st blade elements stagger

Blade.Stagger(:,:,2) = [55 45 15 10;...
                        65 55 25 15;...
                        75 72 70 65]; % 2nd blade elements stagger

Blade.Thickness(:,:,1) = [0.040  0.08 0.10 0.08;...
                          0.015  0.03 0.06 0.02;...
                          0.005  0.015 0.025 0.005]; % 1st blade elements
thickness...0.005   0.015 0.03 0.005]
```

```
Blade.Thickness(:,:,2) = [0.040   0.08 0.10 0.08;...
                          0.015   0.03 0.06 0.02;...
```

```
                          0.005   0.02 0.04 0.005]; % 2nd blade elements
thickness


Blade.Offset    = [0.0 0.35;...        % fraction = 1 would rotate across the
entire passage
                  0.0 0.35;...
                  0.0 0.35];           % Fraction of passage to rotate each
blade element [radius; fraction; fraction... raduis; fraction; fraction...]

Blade.MasterXShift = -0.2;                % Distance to offset all the blades
in the x-direction in inches
Blade.Fillet = 0.000;                     % Fillet radius in inches
Blade.Center = false;
Blade.CentroidCorrection = 0.0;
```

IX. HARDCODEPARAMS.M

```
% Function that generates a hard-coded 'Params'

function Params = HardCodeParams

% Gannon
Params.Air.Cp = 1004;         % Specific Heat Capacity (J/kg-K)
Params.Air.Gam = 1.4;         % Ratio of Specific Heats
Params.Inlet.Po = 101335;     % Inlet pressure (N/m^2)
Params.Inlet.To = 288.15;     % Inlet temperature (K)
Params.PR = 2.0;              % Desired rotor pressure ratio
Params.RPM = 27000;           % Desired 100% rotor speed (RPM)

% Thesis
Params.W = 171.3;             % Specific Weightflow (kg/s-m^2)
Params.rho1 = 1.2235;         % Ambient Density (kg/m^3)
Params.R = 0.287;             % Gas Constant (kJ/kg-K)
%Params.PSI = 0.246;          % Specific Head Ratio Parameter
Params.eta = 0.90;            % Rotor Efficiency
Params.AR = 1.2;              % Aspect Ratio
Params.SA = 50;               % Midspan Stagger Angle (deg)
Params.AVR = 0.9;             % Axial Velocity Ratio
Params.TIR = 0.14351;   % Rotor Tip Inlet Radius (m) Original = 0.1435 m
Params.TER = 0.14351;   % Rotor Tip Exit Radius (m)  Original = 0.1435 m
Params.HER = 0.10668;         % Rotor Hub Exit Radius (m)
Params.OCD = 0.2881376;       % Outer Casing Diameter (m)  Original = 0.28702 m
New value reflects 22 thou TG
Params.OT_axial = 0.03;       % Axial Outlet Hub Transition (m)
Params.OT_radial = 0.01;      % Radial Outlet Hub Transition (m)

% Other
Params.UpstreamAxial = 0.36;    % Upstream Axial Wedge Length (# of OCDs)
Params.DownstreamAxial = 0.25; % Downstream Axial Wedge Length (# of OCDs)
```

X. GEOMGEN.M

```
% m-file to calculate the flow through the Sanger rotor. A still very
% simple analysis
% Note the methods work in SI units
%
% Fields necessary for argument "Blade":
%   PassNo:  Number of blade passages; number of times the calculated blade
profile(s) are repeated around the hub
%   S: Number of sections used to generate the blade
%   P: Number of points that define a blade profile.  One blade profile
actually contains twice this number of points.
%   Heights:  Specifies heights for property inputs
%       Entered as a vertical vector
%   Chord:  Passage chord at "Heights" location
%       One column for each blade element
%   LE:  Blade leading edge shift backwards/forwards at "Heights" as a
fraction of axial chord
%       One column for each blade element
%   Edges:  Blade leading and trailing edge at "Heights"
%       Entered as a three-dimensional array
%       Rows are in the form:  [Leading edge minor axis/Chord | Leading edge
eccentricity | Trailing edge minor axis/Chord | Trailing edge eccentricity]
%       Each row matches with a height in "Heights"
%       Third dimension corresponds to blade element
%   Controls:  Chord control point locations
%       Entered as a horizontal vector
%   Stagger:  Stagger of blade elements
%       Entered as a three-dimensional array
%       Rows match "Heights"
%       Columns match chord control point locations
%       Third dimension corresponds to blade element
%   Thickness:  Blade element thickness
%       Entered as a three-dimensional array
%       Rows match heights
%       Columns match chord control point locations
%       Third dimension corresponds to blade element
%   Offset:  Fraction of passage to rotate each blade element
%       Fraction = 1 would rotate across the entire passage
%       Rows are in the form [radius | fraction]
%       Columns match heights
%   MasterXShift:  Additional distance in inches to shift all blades in the
x-direction
%   Fillet:  Fillet radius of the blade in inches
%   Center:  Boolean specifying whether to center the blade on the hub origin
(true) or align the leading edge with the origin (false) before MasterXShift
is applied
%   CentroidCorrection:  Number between 0 and 1 indicating how much to
%       correct the horizontal shift of the blade section from the centroid
%       0 is no correction, 1 lines up on the main axis
%
% Fields necessary for argument "Params":
%   Air.Cp:  Coefficient of specific heat
%   Air.Gam:  Specific heat ratio of air
%   Inlet.Po:  Standard atmospheric pressure at inlet in Pa
%   Inlet.To:  Standard atmospheric temperature in K
```

```
%   PR:  Design pressure ratio
%   RPM:  Design RPM
%   W:  Specific Weightflow
%   rho1:  Ambient Density
%   R:  Gas Constant in kJ/(kg K)
%   eta:  Rotor Efficiency
%   AR:  Aspect Ratio
%   SA:  Midspan Stagger Angle in degrees
%   AVR:  Axial Velocity Ratio
%   TIR:  Rotor Tip Inlet Radius in meters
%   TER:  Rotor Tip Exit Radius in meters
%   HER:  Rotor Hub Exit Radius in meters
%   OCD:  Outer Casing Diameter in meters
%   OT_axial:  Axial Outlet Hub Transition in meters
%   OT_radial:  Radial Outlet Hub Transition in meters
%   UpstreamAxial:  Upstream axial wedge length in number of casing diameters
%   DownstreamAxial:  Downstream axial wedge length in number of casing
diameters


function [Blade, Wedge] = GeomGen(Blade, Params)



% Constants
%m_dot = 7.5;            % Mass flow rate
%rho   = 1.225;          % Density
%Air.Cp   = 1005;           % Coefficienct of specific heat
%Air.Gam  = 1.4;            % Specific heat ratio of air
Params.Air.R     = Params.Air.Cp*(Params.Air.Gam-1)/Params.Air.Gam; % Gas
constant for air
%Inlet.Po = 101335;         % Standard atmospheric pressure at inlet
%Inlet.To = 288.15;         % Standard atmospheric temperature
Params.Inlet.ho = Params.Air.Cp*Params.Inlet.To;       % Inlet stagnation
enthalpy
%a     = 6;              % Constant for actuator disc
%b     = 150;            % Constant for actuator disc
%PR    = 1.56;           % Design Pressure Ratio
%RPM   = 27085;          % Design RPM
Omega = 2*pi*Params.RPM/(60);  % RPM in Radians per second

% DirectoryName = 'F:\Anthony Gannon\Matlab\TCR\Redesign\STFM\Sanger\';

% Ensuring all dimensions match & backwards compatible with program
Blade.LE       = [Blade.Heights Blade.LE];

% Blade Geometry
[Blade] = BladeGen(Params.PR,Omega,Blade);

%SolidGen(Blade,Wedge,z,r,objApp)

% Set up grid
N_z = [10 10];     % No. of grid points in z direction [upstream downstream]
N_r = 12;          % No. of streamlines.
```

```
% Run hub generation
[Hub.x Hub.y Casing.x Casing.y Params] = SangerMethod(Params);

% Initial values
Stream.psi = (0:(Params.m_dot/(2*pi))/(N_r-
1):Params.m_dot/(2*pi))'*ones(1,sum(N_z)); % Stream function value
psi = Stream.psi;

% Passage geometry
[Stream.z,Stream.r, HubProfile] = Passage(N_z,N_r,Blade,Hub,Casing);

% Gas Wedge to contain blades
[Wedge]  = WedgeGen(Blade,Stream.z,Stream.r);

% Blade grid position, needs to be modified to find actual blade
%z_blade = min(find(Stream.z(1,:)>0));
z_blade = N_z(1) + 1;

% Initial gas properties
ho          = Params.Inlet.ho*ones(size(Stream.r));   % Stagnation enthalpy
Stream.po     = Params.Inlet.Po*ones(size(Stream.r));   % Stagnation pressure
Stream.rho_o = Params.Air.Cp*Stream.po./(Params.Air.R*ho); % Stagnation
density
Stream.rho    = Stream.rho_o;                  % Initial static density, could
modify this to take at least axial velocity into account
p           = Stream.po;                    % Static pressure
h           = ho;                    % Static enthalpy
Stream.s   = zeros(size(Stream.r));        % Entropy

% Initial enthalpy distribution through machine with blade work
%dhr = Omega*(r(:,min(find(z(1,:)>0))).*Cth(:,min(find(z(1,:)>0))) - 0); %
Euler turbomachinery equation
%ho(:,min(find(z(1,:)>0)):end) =ho(:,min(find(z(1,:)>0)):end) +
dhr*ones(1,length(min(find(z(1,:)>0)):size(z,2)));

%figure(1);close;
%figure(1);plot(z',r','+r'); axis equal;

disp('Paused before simulation')
pause(1)

%r_old = r;
%r(2:end-1,:) = r(2:end-1,:)*1.01;
%r   = flipud(r); psi = flipud(psi);

count.Stream = 0;
tol = 1;
Stream.relax = 0.9;
tic

% Conversion to SI units
Stream.z = Stream.z.*0.0254; Stream.r = Stream.r.*0.0254;
Wedge.x   = Wedge.x.*0.0254;  Wedge.y   = Wedge.y.*0.0254; Wedge.z   =
Wedge.z.*0.0254;
%Stream.z = z; Stream.r = r;
```

```
while tol > 1e-5 && count.Stream < 2
    [log(tol)-log(1e-5) count.Stream]

    % Enthalpy change across blade, this will change with streamline shift
    % in a real machine
    Stream.ho   = Params.Inlet.ho*ones(size(Stream.r));                    %
Inlet stagnation enthalpy
    dhr         = ones(N_r,1)*Params.Inlet.ho*(Params.PR^((Params.Air.Gam-
1)/Params.Air.Gam)-1); % Required enthalpy for pressure ratio
    Stream.ho(:,z_blade:end) = Stream.ho(:,z_blade:end) +
dhr*ones(1,length(z_blade:size(Stream.z,2)));

    % Tangential velocities, must also be updated as the r value changes
    % and angular momentum remains constant
    Stream.Cth = ([-zeros(N_r,N_z(1))
(dhr*ones(1,N_z(2))./Omega)./Stream.r(:,z_blade:end)]); % Forced vortex
distribution to obtain desired pressure ratio

    % Ideal Pressures and densities
    Stream.po =
Params.Inlet.Po.*(Stream.ho/Params.Inlet.ho).^(Params.Air.Gam/(Params.Air.Gam
-1));
    rho_o = Params.Air.Cp*Stream.po./(Params.Air.R*Stream.ho);              %
Stagnation density

    % Losses would be subtracted here, recall they are cumulative along a
    % streamline
    [Stream,tol,fred] = StreamGen(Stream,Params.Air,Params.Inlet);

    count.Stream = count.Stream + 1;

end
tol
count.Stream
toc

% Plot streamlines in inches
figure(1);%close;figure(1);
plot(Stream.z'./0.0254,Stream.r'./0.0254,'-b'); %axis equal;
hold on;

% Velocities
figure(3); close; figure(3);
contourf(Stream.z,Stream.r,(sqrt(Stream.Cth.^2+Stream.Cz.^2+Stream.Cr.^2)));
axis equal
colorbar
title('C')

% Axial velocity
figure(4); close; figure(4);
contourf(Stream.z,Stream.r,Stream.Cz); axis equal
colorbar
title('Cz')

% Radial velocity
```

```
figure(5); close; figure(5);
contourf(Stream.z,Stream.r,Stream.Cr); axis equal
colorbar
title('Cr')

% Tangential Velocity
figure(6); close; figure(6);
contourf(Stream.z,Stream.r,Stream.Cth); axis equal
colorbar
title('Cth')

% Inlet, outlet and tangential profiles
figure(7); close; figure(7);
plot(Stream.r(:,1),Stream.Cz(:,1),'-r',Stream.r(:,1),Stream.Cz(:,1),'+r');
hold on % Inlet
plot(Stream.r(:,end),Stream.Cz(:,end),'-
g',Stream.r(:,end),Stream.Cz(:,end),'+g'); % Outlet
%plot(r(:,end),Cth(:,end),'-b',r(:,end),Cth(:,end),'+b'); % Whirl
title('Inlet and outlet velocity profiles')

(Stream.Cz(end,end)-Stream.Cz(1,end))/Stream.Cz(end,end)

%-----------------------------------------------------------------------

% Conversion to SI units
Blade.x   = Blade.x.*0.0254;  Blade.y   = Blade.y.*0.0254; Blade.z   =
Blade.z.*0.0254;
Blade.r = Blade.r.*0.0254;
Blade.Fillet = Blade.Fillet*.0254;

HubProfile.r = HubProfile.r.*0.0254; HubProfile.z = HubProfile.z.*0.0254;

% Close all open SolidWorks files
h1    = NET.addAssembly('C:\Program Files\SolidWorks
Corp\SolidWorks\SolidWorks.Interop.sldworks.dll'); % 2010 version
swApp = SolidWorks.Interop.sldworks.SldWorksClass;
swApp.CloseAllDocuments(true);

disp('Beginning geometry generation...')
SolidWorksGen
BladeHub_Wedge_CutOut
disp('Geometry generation complete.')

% Close all open SolidWorks files if window is not visible.  Will quit the
% program (necessary since the program is completely invisible to the user
% in this state).
if ~(swApp.Visible)
    swApp.CloseAllDocuments(true);
end
```

XI. BLADEGEN.M

```
function [Blade] = BladeGen(PR,Omega,Blade)
% Matlab function file to generate blade profiles for solid generation

% Some useful constants
StationHeight = linspace(Blade.Heights(1,1),Blade.Heights(end,1),Blade.S);
Blade.P       = 2*floor(Blade.P/2); % Ensures an even value

figure(2); close; figure(2)

% Loops to do each section
for ii = 1:Blade.S
    for jj = 1:(size(Blade.LE,2)-1)
        % Blade stack heights
        Blade.y(:,ii,jj) = ones(2*Blade.P,1)*StationHeight(ii);

        % Fraction of chord length that blade it slid along its chord
        ChordShift = interp1(Blade.LE(:,1), Blade.LE(:,1+jj),
StationHeight(ii), 'spline');

        % Unit length blade sections are generated (pass: section height,
        % Blade element no
        [~,Blade.z(:,ii,jj),Blade.x(:,ii,jj)] =
BladeSect(Blade,StationHeight(ii),jj,ChordShift);

        % Blade chord scaled to specified overall length
        ChordLength    = interp1(Blade.Heights, Blade.Chord(:,jj),
StationHeight(ii), 'spline');

        % Blade lengths are scaled to overall desired length
        Blade.z(:,ii,jj) = ChordLength*Blade.z(:,ii,jj);
        Blade.x(:,ii,jj) = ChordLength*Blade.x(:,ii,jj);

        % This should be done once both blades have been made.
        % Overall axial chord is stored
        AxialChord = Blade.z(Blade.P+1,ii,jj) - Blade.z(1,ii,jj);
        % Leading edge shift

        % Plot of blade chordline
        figure(2)

        % Overplot onto streamlines of unstaggered blades
        figure(1); hold on
        plot(Blade.z(:,ii,jj),Blade.y(:,ii,jj),'.g')

        % Blade outline is plotted and handle assigned for rotations
        figure(2); hold on; axis equal
        f(ii,jj) =
plot3(Blade.z(:,ii,jj),Blade.y(:,ii,jj),Blade.x(:,ii,jj),'-k');      hold on;

        % Blade is rotated around the rotational axis to set splitter
positions
```

```
        %[ii jj
interp1(Blade.Offset(:,1),Blade.Offset(:,jj+1),Blade.y(1,ii))*(360/Blade.Pass
No)/(size(Blade.LE,2)-1)]
        temp.Offset =
interp1(Blade.Heights,Blade.Offset(:,jj),StationHeight(ii))*(360/Blade.PassNo
);

        %rotate(f(ii,jj),[0    0],-(jj-1)*(360/Blade.PassNo)/(size(Blade.LE,2)-
1),[0 0 0]);
        rotate(f(ii,jj),[0 0],temp.Offset,[0 0 0]);

        % Cartesian coordinates
        Blade.z(:,ii,jj) = get(f(ii,jj),'XData');
        Blade.y(:,ii,jj) = get(f(ii,jj),'YData');
        Blade.x(:,ii,jj) = get(f(ii,jj),'ZData');

        % Polar coordinates
        Blade.r(:,ii,jj)   = sqrt(Blade.x(:,ii,jj).^2+Blade.y(:,ii,jj).^2);
        Blade.th(:,ii,jj) = atan2(Blade.x(:,ii,jj),Blade.y(:,ii,jj));

        % Overplot onto streamlines
        figure(1); hold on
        plot(Blade.z(:,ii,jj),Blade.y(:,ii,jj),'.k')

        % Spreadsheets are written
        %filename = ['BladeSect_' num2str(ii) '_' num2str(jj) '.xls' ];
        %delfile = ['! del ' filename]; % Delete file if it exists
        %eval(delfile)
        %xlswrite(filename,[Blade.z(:,ii,jj) Blade.y(:,ii,jj)
Blade.x(:,ii,jj)])

        % Text files are written
        %filename = ['BladeSect_' num2str(ii) '_' num2str(jj) '.txt' ];
        %delfile = ['! del ' filename]; % Delete file if it exists
        %eval(delfile)
        %dlmwrite(filename, [[Blade.z(:,ii,jj); Blade.z(1,ii,jj)]
[Blade.y(:,ii,jj); Blade.y(1,ii,jj)] [Blade.x(:,ii,jj); Blade.x(1,ii,jj)]],
'delimiter', '\t')

    end % for jj
end % for ii

figure(2)
xlabel('z'); ylabel('y'); zlabel('x');
grid on

% Line up the leading edge of the primary blade with the origin unless
Blade.Center is true
% Determine the distance between the base of the blade and the origin
% This is the same as the absolute value of the extreme leading edge at the
base of the blade
if ~Blade.Center
    alignShift = abs(min(Blade.z(:,1,1)));
else
    alignShift = 0;
```

```
end

% Add on the master x offset as well
Blade.z = Blade.z + alignShift + Blade.MasterXShift;

for ii = 1:Blade.S
    for jj = 1:(size(Blade.LE,2)-1)
        set(f(ii,jj),'XData',Blade.z(:,ii,jj))
    end
end


for jj = 1:(size(Blade.LE,2)-1)
    plot3(Blade.z(:,:,jj)',Blade.y(:,:,jj)',Blade.x(:,:,jj)','-k')
end

end
```

XII. BLADESECT.M

```
function [ Blade,X,Y ] = BladeSect( Blade, StationHeight, BladeElement,
ChordShift )
%BladeSect Function for producing individual blades
%   Generalized blade fitting function.
%
% Input variables
% Blade:          The general storage variable of blade properties
% StationHeight: Height of generated blade section for interpolation
% BladeElement:  Number of particular blade element being generated

TempY        = Blade.Heights';
TempX        = Blade.Controls;
TempZ        = Blade.Stagger(:,:,BladeElement);
TempYI       = StationHeight*ones(1,length(Blade.Controls));
TempXI       = Blade.Controls;
Chord.Angle = [Blade.Controls' (interp2(TempX,TempY,TempZ,TempXI,TempYI))'];

TempZ              = Blade.Thickness(:,:,BladeElement);
Chord.ThickCoarse = [Blade.Controls'
(interp2(TempX,TempY,TempZ,TempXI,TempYI))'];
Chord.ThickCoarse = [Chord.ThickCoarse(:,1) 0.5*Chord.ThickCoarse(:,2)]; %
Thickness is halved

Chord.ThickLE   =
interp1(Blade.Heights,Blade.Edges(:,1,BladeElement),StationHeight);
Chord.EccentLE =
interp1(Blade.Heights,Blade.Edges(:,2,BladeElement),StationHeight);
Chord.ThickTE   =
interp1(Blade.Heights,Blade.Edges(:,3,BladeElement),StationHeight);
Chord.EccentTE =
interp1(Blade.Heights,Blade.Edges(:,4,BladeElement),StationHeight);

%Chord.ThickCoarse = [Chord.ThickCoarse; 1 0] % Addition of leading and
trailing point

Chord.Slope       = [Chord.Angle(:,1) tand(Chord.Angle(:,2))];         % Slope
rather than angle
Chord.X           = (fliplr(0.5+0.5*cos(pi*(0:Blade.P)/Blade.P)))';     % x-
coords with end concentration

Chord.Stagger     = (max(Chord.Angle(:,2)) + min(Chord.Angle(:,2)))/2; % Mean
stagger angle

pp1           = spapi(3,Chord.Slope(:,1),Chord.Slope(:,2)); % Spline fit of
slopes
Chord.SlopeX = fnval(pp1,Chord.X);                          % Slope at many
point
Chord.Y       = fnval(fnint(pp1),Chord.X);                  % Integration of
slope
%Chord.Y       = Chord.Y - Chord.Y(Chord.X==0.5);           % Blade is stacked
around center for now
```

```
% Blade is scaled to a unit length
```

```
Chord.XUnit = Chord.X./sqrt((Chord.X(end)-Chord.X(1))^2+(Chord.Y(end)-
Chord.Y(1))^2);
Chord.YUnit = Chord.Y./sqrt((Chord.X(end)-Chord.X(1))^2+(Chord.Y(end)-
Chord.Y(1))^2);

% Leading and trailing edge ellipse constants
Ellipse.b_LE    = 0.5*Chord.ThickLE;              % Classical ellipse
equation b
Ellipse.b_TE    = 0.5*Chord.ThickTE;              % Classical ellipse
equation b
Ellipse.a_LE    = Chord.EccentLE*Ellipse.b_LE;    % Classical ellipse
equation a
Ellipse.a_TE    = Chord.EccentTE*Ellipse.b_TE;    % Classical ellipse
equation a
Ellipse.Xc_LE   = Ellipse.a_LE;                   % Classical ellipse
equation Xc
Ellipse.Xc_TE   = 1-Ellipse.a_TE;                 % Classical ellipse
equation Xc

% All leading edge ellipse points (makes finding blend point simpler)
temp.X_LE        = Chord.X(Chord.X<(2*Ellipse.a_LE));
% X points
temp.Y_LE        = Ellipse.b_LE*sqrt(1-((temp.X_LE-
Ellipse.Xc_LE).^2)./(Ellipse.a_LE^2)); % Y points

% All Trailing edge ellipse points (makes finding blend point simpler)
temp.X_TE        = Chord.X(Chord.X>(1-2*Ellipse.a_TE));
% X points
temp.Y_TE        = Ellipse.b_TE*sqrt(abs(1-((temp.X_TE-
Ellipse.Xc_TE).^2)./(Ellipse.a_TE^2))); % Y points

% Initial Leading and trailing edge to blade surface transition points,
% these will needed to be iterated to get the correct points
Chord.XLE        = Chord.EccentLE*0.5*Chord.ThickLE;    % Leading X
Chord.XTE        = 1-Chord.EccentTE*0.5*Chord.ThickTE; % Trailing X

% Leading and trailing zero thickness points are superfluous
Chord.ThickCoarse = Chord.ThickCoarse(Chord.ThickCoarse(:,1)>Chord.XLE &
Chord.ThickCoarse(:,1)<Chord.XTE,:);

NotDone = 1; LE_Length = 0; TE_Length = 0;
while NotDone
    %[LE_Length length(Chord.X(Chord.XLE>Chord.X)) TE_Length
length(Chord.X(Chord.X>Chord.XTE))]
    % Doing it this way ensures it gets done one more time
    if LE_Length == length(Chord.X(Chord.XLE>Chord.X)) && TE_Length ==
length(Chord.X(Chord.X>Chord.XTE))
        NotDone = 0;
    end

    % Length check to see if more iterations are needed
    LE_Length = length(Chord.X(Chord.X<Chord.XLE));
    TE_Length = length(Chord.X(Chord.X>Chord.XTE));

    % Y transition points
```

```
    Chord.YLE        = Ellipse.b_LE*sqrt(1-((Chord.XLE-
Ellipse.Xc_LE).^2)./(Ellipse.a_LE^2)); % Leading Y
    Chord.YTE        = Ellipse.b_TE*sqrt(1-((Chord.XTE-
Ellipse.Xc_TE).^2)./(Ellipse.a_TE^2)); % Trailing Y

    % Control points are updated to include the transition points
    Chord.ThickCoarse2 = [Chord.XLE Chord.YLE; Chord.ThickCoarse; Chord.XTE
Chord.YTE];

    temp.X_top       = Chord.X(Chord.X>Chord.XLE & Chord.X<Chord.XTE); % X
points on the top surface
    % Polynomial fit of top surface through control points
    pptop            =
spapi(3,Chord.ThickCoarse2(:,1),Chord.ThickCoarse2(:,2)); % Spline fit
    temp.Y_top       = fnval(pptop,[temp.X_top]);

    % Slopes of blade elements
    temp.dX_LE       = fnval(fnder(spapi(3,temp.X_LE,temp.Y_LE),1),temp.X_LE);
% Leading edge
    temp.dX_TE       = fnval(fnder(spapi(3,temp.X_TE,temp.Y_TE),1),temp.X_TE);
% Trailing edge
    temp.dX_top      =
fnval(fnder(spapi(3,temp.X_top,temp.Y_top),1),temp.X_top); % Top surface edge

    % New transition point is found by matching the slopes
    Chord.XLE        = interp1(temp.dX_LE,temp.X_LE,temp.dX_top(1));
    Chord.XTE        = interp1(temp.dX_TE,temp.X_TE,temp.dX_top(end));

end % While NotDone

% Thickness points on fine grid are assigned
Chord.Thickness = [temp.Y_LE(temp.X_LE<temp.X_top(1)); temp.Y_top;
temp.Y_TE(temp.X_TE>temp.X_top(end))];

% Final blade shape is defined with the thickness distributed about the
% chord and the local slope
X = Chord.XUnit-Chord.Thickness.*sin(atan(Chord.SlopeX));
Y = Chord.YUnit+Chord.Thickness.*cos(atan(Chord.SlopeX));
X = [X; flipud(Chord.XUnit(2:(end-1))+Chord.Thickness(2:(end-
1)).*sin(atan(Chord.SlopeX(2:(end-1)))))];
Y = [Y; flipud(Chord.YUnit(2:(end-1))-Chord.Thickness(2:(end-
1)).*cos(atan(Chord.SlopeX(2:(end-1)))))];

% Blade is moved to centriod of area
[ geom, iner, cpmo ] = polygeom( X, Y );
X = X - geom(2);                    Y = Y - geom(3);
Chord.XUnit = Chord.XUnit -geom(2); Chord.YUnit = Chord.YUnit -geom(3);

% Blade is shifted backwards of forwards using a parametric approach
XShift = [Chord.XUnit(1) 0 Chord.XUnit(end)]; % X[lead AreaCentriod trail]
YShift = [Chord.YUnit(1) 0 Chord.YUnit(end)]; % Y[lead AreaCentriod trail]

xy = [XShift; YShift];                  % Grouping of vectors
df = diff(xy,1,2);                      % Difference of vectors
t = cumsum([0, sqrt([1 1]*(df.*df))]); % Pythag to get cumulative distance
```

```
t = t - t(2);                            % Centering about centroid
XY = fnval(csapi(t,xy),ChordShift);      % New Centroid

%X            = X + XY(1);           Y            = Y + XY(2);
%Chord.XUnit = Chord.XUnit + XY(1); Chord.YUnit = Chord.YUnit + XY(2);
Y            = Y + ChordShift;
Chord.YUnit = Chord.YUnit + ChordShift;
%ChordShift
%XY
%disp('paused at chordshift')
%pause

% Plot of slope
figure(21); close; figure(21);
plot(Chord.Slope(:,1),Chord.Slope(:,2),'or'); grid on; hold on
plot(Chord.X,Chord.SlopeX,'-+g');
xlabel('x'); ylabel('Slope of chord about stagger'); title('x vs Slope')

% Plot of camber line to satisfy the angle
figure(22); close; figure(22);
hh = plot(Chord.XUnit,Chord.YUnit,'-+g'); grid on; hold on; axis equal
plot(X,Y,'-k') % Blade shape
plot(X-ChordShift,Y,'-k') % Blade shape before shift
xlabel('x'); ylabel('Chord about stagger'); title('x vs Chord')

figure(23); close; figure(23);
plot(Chord.ThickCoarse2(:,1),Chord.ThickCoarse2(:,2),'or'); grid on; hold on
%plot([Chord.X],Chord.ThickFine,'-+g');
plot(temp.X_top,temp.Y_top,'-+g');
plot(temp.X_LE,temp.Y_LE,'-ob'); % Complete nose ellipse
plot(temp.X_TE,temp.Y_TE,'-ob'); % Complete tail ellipse
plot(Chord.X,Chord.Thickness,'-ok')
xlabel('x'); ylabel('Thickness of blade'); title('x vs Thickness')
axis equal

% Line of leading and trailing edge ellipses
temp.axis = axis;
% LE
plot([Chord.XLE Chord.XLE],[temp.axis(3) temp.axis(4)],'-r')
plot(Chord.XLE,Chord.YLE,'or')
% TE
plot([Chord.XTE Chord.XTE],[temp.axis(3) temp.axis(4)],'-r')
plot(Chord.XTE,Chord.YTE,'or')

% Slopes of leading ellipses and top of blade
figure(24); close; figure(24);
plot(temp.X_LE,temp.dX_LE,'-ob'); hold on; grid on
plot(temp.X_TE,temp.dX_TE,'-ob');
plot(temp.X_top,temp.dX_top,'-+g');
xlabel('x'); ylabel('Slope'); title('x vs Slope (LE, TE, Top Surface)')

figure(22)
%disp('Paused in Blade Sect')
%pause
polygeom
```

XIII. **POLYGEOM.M**

```
function [ geom, iner, cpmo ] = polygeom( x, y )
%POLYGEOM Geometry of a planar polygon
%
%   POLYGEOM( X, Y ) returns area, X centroid,
%   Y centroid and perimeter for the planar polygon
%   specified by vertices in vectors X and Y.
%
%   [ GEOM, INER, CPMO ] = POLYGEOM( X, Y ) returns
%   area, centroid, perimeter and area moments of
%   inertia for the polygon.
%   GEOM = [ area   X_cen   Y_cen   perimeter ]
%   INER = [ Ixx    Iyy     Ixy     Iuu     Ivv     Iuv ]
%     u,v are centroidal axes parallel to x,y axes.
%   CPMO = [ I1     ang1    I2      ang2    J ]
%     I1,I2 are centroidal principal moments about axes
%         at angles ang1,ang2.
%     ang1 and ang2 are in radians.
%     J is centroidal polar moment.  J = I1 + I2 = Iuu + Ivv

% H.J. Sommer III - 02.05.14 - tested under MATLAB v5.2
%
% code available at:
%    http://www.me.psu.edu/sommer/me562/polygeom.m
% derivation of equations available at:
%    http://www.me.psu.edu/sommer/me562/polygeom.doc
%
% sample data
% x = [ 2.000  0.500  4.830  6.330 ]';
% y = [ 4.000  6.598  9.098  6.500 ]';
% 3x5 test rectangle with long axis at 30 degrees
% area=15, x_cen=3.415, y_cen=6.549, perimeter=16
% Ixx=659.561, Iyy=201.173, Ixy=344.117
% Iuu=16.249, Ivv=26.247, Iuv=8.660
% I1=11.249, ang1=30deg, I2=31.247, ang2=120deg, J=42.496
%
% H.J. Sommer III, Ph.D., Professor of Mechanical Engineering, 337 Leonhard
Bldg
% The Pennsylvania State University, University Park, PA  16802
% (814)863-8997  FAX (814)865-9693  hjs1@psu.edu  www.me.psu.edu/sommer/

% begin function POLYGEOM

% check if inputs are same size
if ~isequal( size(x), size(y) ),
  error( 'X and Y must be the same size');
end

% number of vertices
[ x, ns ] = shiftdim( x );
[ y, ns ] = shiftdim( y );
[ n, c ] = size( x );

% temporarily shift data to mean of vertices for improved accuracy
```

```
xm = mean(x);
ym = mean(y);
x = x - xm*ones(n,1);
y = y - ym*ones(n,1);

% delta x and delta y
dx = x( [ 2:n 1 ] ) - x;
dy = y( [ 2:n 1 ] ) - y;

% summations for CW boundary integrals
A = sum( y.*dx - x.*dy )/2;
Axc = sum( 6*x.*y.*dx -3*x.*x.*dy +3*y.*dx.*dx +dx.*dx.*dy )/12;
Ayc = sum( 3*y.*y.*dx -6*x.*y.*dy -3*x.*dy.*dy -dx.*dy.*dy )/12;
Ixx = sum( 2*y.*y.*y.*dx -6*x.*y.*y.*dy -6*x.*y.*dy.*dy ...
          -2*x.*dy.*dy.*dy -2*y.*dx.*dy.*dy -dx.*dy.*dy.*dy )/12;
Iyy = sum( 6*x.*x.*y.*dx -2*x.*x.*x.*dy +6*x.*y.*dx.*dx ...
          +2*y.*dx.*dx.*dx +2*x.*dx.*dx.*dy +dx.*dx.*dx.*dy )/12;
Ixy = sum( 6*x.*y.*y.*dx -6*x.*x.*y.*dy +3*y.*y.*dx.*dx ...
          -3*x.*x.*dy.*dy +2*y.*dx.*dx.*dy -2*x.*dx.*dy.*dy )/24;
P = sum( sqrt( dx.*dx +dy.*dy ) );

% check for CCW versus CW boundary
if A < 0,
  A = -A;
  Axc = -Axc;
  Ayc = -Ayc;
  Ixx = -Ixx;
  Iyy = -Iyy;
  Ixy = -Ixy;
end

% centroidal moments
xc = Axc / A;
yc = Ayc / A;
Iuu = Ixx - A*yc*yc;
Ivv = Iyy - A*xc*xc;
Iuv = Ixy - A*xc*yc;
J = Iuu + Ivv;

% replace mean of vertices
x_cen = xc + xm;
y_cen = yc + ym;
Ixx = Iuu + A*y_cen*y_cen;
Iyy = Ivv + A*x_cen*x_cen;
Ixy = Iuv + A*x_cen*y_cen;

% principal moments and orientation
I = [ Iuu  -Iuv ;
     -Iuv   Ivv ];
[ eig_vec, eig_val ] = eig(I);
I1 = eig_val(1,1);
I2 = eig_val(2,2);
ang1 = atan2( eig_vec(2,1), eig_vec(1,1) );
ang2 = atan2( eig_vec(2,2), eig_vec(1,2) );
```

```
% return values
geom = [ A   x_cen   y_cen   P ];
iner = [ Ixx Iyy    Ixy Iuu    Ivv   Iuv ];
cpmo = [ I1  ang1   I2  ang2   J ];

% end of function POLYGEOM
```

XIV. SANGERMETHOD.M

```
% Code for Sanger's Design Methodology for NPS Transonic Compressor

function [X_hubfinal Y_hubfinal X_casingfinal Y_casingfinal Params] =
SangerMethod(Params)

% ******************************************
% Collect required inputs - auto default to
% Sanger's Final Design Compromise Parameters
% ******************************************
if 0    % Old method of entering parameters
% Ambient Pressure (P1)
P01 = input('Enter a value for Ambient Pressure - P01 (N/m^2): ')
if isempty(P01)
    P01 = 101325
end
% Ambient Temperature (T1)
T01 = input('Enter a value for Ambient Temperature - T01 (K): ')
if isempty(T01)
    T01 = 288
end
% Ambient Density (rho1)
rho1 = input('Enter a value for Ambient Density - rho1 (kg/m^3): ')
if isempty(rho1)
    rho1 = 1.2235
end
% 100% Rotor Speed (RPM)
RPM = input('Enter a value for 100% Rotor Speed - RPM (RPM): ')
if isempty(RPM)
    RPM = 27085
end
% Specific Heat Capacity (Cp)
Cp = input('Enter a value for Specific Heat Capacity - Cp (kJ/kg-K): ')
if isempty(Cp)
    Cp = 1.004
end
% Ratio of Specific Heats (Gamma)
Gamma = input('Enter a value for Ratio of Specific Heats - Gamma: ')
if isempty(Gamma)
    Gamma = 1.4
end
% Gas Constant (R)
R = input('Enter a value for the Gas Constant - R (kJ/kg-K): ')
if isempty(R)
    R = 0.287
end
% Specific Head Ratio Parameter (PSI)
PSI = input('Enter a value for Specific Head Ratio Parameter - PSI: ')
if isempty(PSI)
    PSI = 0.246
end
% Rotor Pressure Ratio (PR)
PR = input('Enter a value for Rotor Pressure Ratio - PR: ')
```

```
if isempty(PR)
    PR = 1.58
```

```
end
% Rotor Efficiency (eta)
eta = input('Enter a value for Rotor Efficiency - eta (0.XX): ')
if isempty(eta)
    eta = .92
end
% Aspect Ratio (AR)
AR = input('Enter a value for Aspect Ratio - AR: ')
if isempty(AR)
    AR = 1.2
end
% Midspan Stagger Angle (SA)
SA = input('Enter a value for midspan Stagger Angle - SA (deg): ')
if isempty(SA)
    SA = 50
end
% Specific Weightflow (W)
W = input('Enter a value for Specific Weightflow - W (kg/s-m^2): ')
if isempty(W)
    W = 171.3
end
% Axial Velocity Ratio (AVR)
AVR = input('Enter a value for Axial Velocity Ratio - AVR (Vz2/Vz1): ')
if isempty(AVR)
    AVR = 0.9
end
% Rotor Tip Inlet Radius (TIR)
TIR = input('Enter a value for Rotor Tip Inlet Radius - TIR (m): ')
if isempty(TIR)
    TIR = 0.1397
end
% Rotor Tip Exit Radius (TER)
TER = input('Enter a value for Rotor Tip Exit Radius - TER (m): ')
if isempty(TER)
    TER = 0.1397
end
% Rotor Hub Exit Radius (HER)
HER = input('Enter a value for Rotor Hub Exit Radius - HER (m): ')
if isempty(HER)
    HER = 0.0884
end
% Outer Casing Diameter (OCD)
OCD = input('Enter a value for Outer Casing Diameter - OCD (m): ')
if isempty(OCD)
    OCD = 0.2881376 %Original value: 0.28702 m  New value has 22 thou TG
incorporated
end
% Spinner Parabola Vertex-to-Focus Distance (p)
p = input('Enter a value for Spinner Parabola Vertex-to-Focus Distance - p:
')
if isempty(p)
    p = 0.0184
    %p = 0.015884 (Sanger-derived value)
end
end % if 0
% ********************
% Calculate Parameters
```

```
% ********************

% Calculate Temperature Ratio (TR)
TR = ((Params.PR^((Params.Air.Gam - 1)/(Params.Air.Gam))-1)/Params.eta)+1;
disp(['Temperature Ratio - TR = ', num2str(TR)])

% Calculate Area Ratio (ARAT)
ARAT = Params.PR/TR*Params.AVR;
disp(['Area Ratio - ARAT = ', num2str(ARAT)])

% Calculate Rotor Exit Area (A2)
A2 = (Params.TER^2-Params.HER^2)*pi;
disp(['Rotor Exit Area - A2 (m^2) = ', num2str(A2)])

% Calculate Rotor Inlet Area (A1)
A1 = A2*ARAT;
disp(['Rotor Inlet Area - A1 (m^2) = ', num2str(A1)])

% Calculate Mass Flow Rate (mdot)
Params.m_dot = Params.W*A1;
disp(['Mass Flow Rate - mdot (kg/s) = ', num2str(Params.m_dot)])

% Calculate Total Inlet Area (TIA)
TIA = pi*Params.TIR^2;
disp(['Total Inlet Area - TIA (m^2) = ', num2str(TIA)])

% Calculate Total Exit Area (TEA)
TEA = pi*Params.TER^2;
disp(['Total Exit Area - TEA (m^2) = ', num2str(TEA)])

% Calculate Hub Inlet Radius (HIR)
HIR = ((TIA - A1)/pi)^0.5;
disp(['Hub Inlet Radius - HIR (m) = ', num2str(HIR)])

% DISPLAY Hub Exit Radius (HER)
disp(['Hub Exit Radius - HER (m) = ', num2str(Params.HER)])

% Calculate Inlet Blade Height (IBH)
IBH = Params.TIR - HIR;
disp(['Inlet Blade Height - IBH (m) = ', num2str(IBH)])

% Calculate Exit Blade Height (EBH)
EBH = Params.TER - Params.HER;
disp(['Exit Blade Height - EBH (m) = ', num2str(EBH)])

% Calculate Average Blade Height (ABH)
ABH = (IBH+EBH)/2;
disp(['Average Blade Height - ABH (m) = ', num2str(ABH)])

% Calculate Average Blade Chord (ABC)
ABC = ABH/Params.AR;
disp(['Average Blade Chord - ABC (m) = ', num2str(ABC)])

% Calculate Average Axial Blade Chord (AABC)
```

```
AABC = ABC*cos(Params.SA*pi/180);
disp(['Average Axial Blade Chord - AABC (m) = ', num2str(AABC)])

% Calculate Ramp Angle (RA)
RA = atan((Params.HER - HIR)/AABC)*180/pi;
disp(['Ramp Angle - RA (deg) = ', num2str(RA)])

% Calculate Ramp Slope (RS)
RS = (Params.HER-HIR)/AABC;

% Calculate Power Required To Drive Rotor (PowerkW & Powerhp)
PowerkW =
((Params.m_dot*(Params.Air.Cp/1000)*Params.Inlet.To)/Params.eta)*(Params.PR^(
(Params.Air.Gam-1)/Params.Air.Gam)-1);
disp(['Power Required To Drive Rotor - PowerkW (kW) = ', num2str(PowerkW)])
Powerhp = PowerkW/0.746;
disp(['Power Required To Drive Rotor - Powerhp (hp) = ', num2str(Powerhp)])

% Calculate initial estimate of Inlet Absolute Velocity (Cz1)
Cz1 = Params.m_dot/(Params.rho1*A1);
disp(['Initial Estimate of Inlet Absolute Velocity - Cz1 (m/s) = ',
num2str(Cz1)])

% Calculate initial estimate of Inlet Speed of Sound (a1)
a1 = (Params.Air.Gam*Params.R*1000*Params.Inlet.To)^0.5;
disp(['Initial estimate of Inlet Speed of Sound - a1 (m/s) = ', num2str(a1)])

% Calculate initial estimate of Inlet Mach Number (M1)
M1 = Cz1/a1;
disp(['Initial estimate of Inlet Mach Number - M1 = ', num2str(M1)])

% ***************************
% Inlet Mach Number Iteration
% ***************************

% Calculate Total Density/Static Density Ratio (rhorat)
rhorat = (1+((Params.Air.Gam-1)/2)*M1^2)^(1/(Params.Air.Gam-1));

% Calculate Static Density (rhostat)
rhostat = Params.rho1/rhorat;

% Calculate Total Temp/Static Temp Ratio (temprat)
temprat = 1+((Params.Air.Gam-1)/2)*M1^2;

% Calculate Static Temperature (tempstat)
tempstat = Params.Inlet.To/temprat;

% Calculate revised Inlet Speed of Sound (Reva1)
Reva1 = (Params.Air.Gam*Params.R*1000*tempstat)^0.5;

% Calculate revised Inlet Absolute Velocity (ConvCz1)
ConvCz1 = Params.m_dot/(rhostat*A1);

% Calculate revised Inlet Mach Number (ConvM1)
```

```
ConvM1 = ConvCz1/Reva1;

while abs(ConvM1 - M1) < 0.001
   % Calculate Total Density/Static Density Ratio (rhorat)
   rhorat = (1+((Params.Air.Gam-1)/2)*ConvM1^2)^(1/(Params.Air.Gam-1));

   % Calculate Static Density (rhostat)
   rhostat = Params.rho1/rhorat;

   % Calculate Total Temp/Static Temp Ratio (temprat)
   temprat = 1+((Params.Air.Gam-1)/2)*ConvM1^2;

   % Calculate Static Temperature (tempstat)
   tempstat = Params.Inlet.To/temprat;

   % Calculate revised Inlet Speed of Sound (a1)
   Reva1 = (Params.Air.Gam*Params.R*1000*tempstat)^0.5;

   % Calculate revised Inlet Absolute Velocity (ConvCz1)
   ConvCz1 = Params.m_dot/(rhostat*A1);

   % Calculate revised Inlet Mach Number (ConvM1)
   ConvM1 = ConvCz1/Reva1;
end

disp(['Converged Inlet Absolute Velocity - ConvCz1 (m/s)= ',
num2str(ConvCz1)])
disp(['Converged Inlet Mach Number - ConvM1 = ', num2str(ConvM1)])

% Calculate Inlet Tip Velocity (Utip1)
RPM1 = Params.RPM/60*2*pi;
Utip1 = RPM1*Params.TIR;
disp(['Inlet Tip Velocity - Utip1 (m/s) = ', num2str(Utip1)])

% Calculate Inlet Relative Velocity (W1)
Wz1 = ConvCz1;
W1 = sqrt((Wz1^2) + (Utip1^2));
disp(['Inlet Relative Velocity - W1 (m/s) = ', num2str(W1)])

% Calculate Inlet Tip Relative Flow Angle (Beta1)
Beta1 = atan(Utip1/ConvCz1)*180/pi;
disp(['Inlet Tip Relative Flow Angle - Beta1 (deg) = ', num2str(Beta1)])

% *************************************
% Generate Inlet Velocity Triangle Plot
% *************************************

figure (1)

% Plot converged Inlet Absolute Velocity vector (RevCz1)
quiver(0,0,ConvCz1,0,0)
axis([-50 200 -500 500])
hold on
% Plot Inlet Tip Velocity vector (Utip1)
```

```
quiver (ConvCz1,0,0,Utip1,0)
hold on
% Plot Inlet Relative Velocity vector (W1)
quiver (0,0,ConvCz1,Utip1, 0)
title('Inlet Velocity Triangle');
xlabel('m/s');
ylabel('m/s');

% ***********************************************
% Incorporate Shock Losses
% Assume worse case: Normal Shock in blade passage
% ***********************************************

% Calculate Inlet Blade Tip Mach Number (Mtip1)
Mtip1 = sqrt(Utip1^2+ConvCz1^2)/Reva1;
disp(['Inlet Blade Tip Mach Number - Mtip1 = ', num2str(Mtip1)])

% Calculate downstream Blade Tip Mach Number (Mtip2)
NumMtip2 = (Mtip1^2+(2/(Params.Air.Gam-1)));
DenomMtip2 = (((2*Params.Air.Gam)/(Params.Air.Gam-1)*(Mtip1^2))-1);
Mtip2 = sqrt(NumMtip2/DenomMtip2);
disp(['Downstream Blade Tip Mach Number - Mtip2 = ', num2str(Mtip2)])

% Calculate Normal Shock Static Pressure Ratio (P2_P1)
P2_P1 = (1+Params.Air.Gam*Mtip1^2)/(1+Params.Air.Gam*Mtip2^2);
disp(['Normal Shock Static Pressure Ratio - P2_P1 = ', num2str(P2_P1)])

% Calculate Normal Shock Static Temp Ratio (T2_T1)
NumT2_T1 = 1+((Params.Air.Gam-1)/2)*(Mtip1^2);
DenomT2_T1 = 1+((Params.Air.Gam-1)/2)*(Mtip2^2);
T2_T1 = NumT2_T1/DenomT2_T1;
disp(['Normal Shock Static Temp Ratio - T2_T1 = ', num2str(T2_T1)])

% Calculate Normal Shock Total Pressure Ratio (Pt2_Pt1)
Pt2_Pt1 = ((P2_P1)*((DenomT2_T1/NumT2_T1)^(Params.Air.Gam/(Params.Air.Gam-
1))));
disp(['Normal Shock Total Pressure Ratio - Pt2_Pt1 = ', num2str(Pt2_Pt1)])

% Calculate downstream Absolute Axial Velocity Component (Cz2)
Cz2 = ConvCz1*Params.AVR;
disp(['Downstream Axial Velocity Component - Cz2 (m/s) = ', num2str(Cz2)])

% Calculate T1
T1 = Params.Inlet.To/(1+((Params.Air.Gam-1)/2)*(Mtip1^2));

% Calculate T2
T2 = T2_T1*T1;

% Calculate downstream Total Absolute Velocity (C2)
C2 = Mtip2*sqrt(Params.Air.Gam*Params.R*1000*T2);
disp(['Downstream Total Absolute Velocity - C2 (m/s) = ', num2str(C2)])

% Calculate downstream Absolute Tangential Velocity Component (CTheta2)
CTheta2 = sqrt((C2^2)-(Cz2^2));
```

```
disp(['Downstream Absolute Tangential Velocity Component - CTheta2 (m/s) = ',
num2str(CTheta2)])

% Calculate downstream Relative Tangential Velocity Component (WTheta2)
WTheta2 = Utip1 - CTheta2;
disp(['Downstream Relative Tangential Velocity Component - WTheta2 (m/s) = ',
num2str(WTheta2)])

% Calculate downstream Relative Axial Velocity Component (Wz2)
Wz2 = Cz2;
disp(['Downstream Relative Axial Velocity Component - Wz2 (m/s) = ',
num2str(Wz2)])

% Calculate downstream Total Relative Velocity (W2)
W2 = sqrt((WTheta2^2)+(Wz2)^2);
disp(['Downstream Total Relative Velocity - W2 (m/s) = ', num2str(W2)])

% Calculate downstream Relative Flow Angle (Beta2)
Beta2 = atan(WTheta2/Wz2)*180/pi;
disp(['Downstream Relative Flow Angle - Beta2 (deg) = ', num2str(Beta2)])

% Calculate overall Stage Flow Turning (BetaStage)
BetaStage = Beta1 - Beta2;
disp(['Overall Stage Flow Turning - BetaStage (deg) = ', num2str(BetaStage)])

% ************************************
% Generate Outlet Velocity Triangle Plot
% ************************************

figure (8)

% Plot Downstream Absolute Velocity vector (C2)
quiver(0,0,Cz2,-CTheta2,0)
axis([-50 200 -500 500])
hold on
% Plot Downstream Tip Velocity vector (Utip1)
quiver(Cz2,-CTheta2,0,Utip1,0)
hold on
% Plot Downstream Relative Velocity vector (W2)
quiver (0,0,Wz2,WTheta2,0)

title('Outlet Velocity Triangle');
xlabel('m/s');
ylabel('m/s');

% **************************************
% Generate Combined Velocity Triangle Plot
% **************************************

figure (3)

% Plot converged Inlet Absolute Velocity vector (RevCz1)
quiver(0,0,ConvCz1,0,0)
axis([-50 200 -500 500])
```

```
hold on
% Plot Inlet Tip Velocity vector (Utip1)
quiver (ConvCz1,0,0,Utip1,0)
hold on
% Plot Inlet Relative Velocity vector (W1)
quiver (0,0,ConvCz1,Utip1, 0)
% Plot Downstream Absolute Velocity vector (C2)
quiver(0,0,Cz2,-CTheta2,0)
hold on
% Plot Downstream Tip Velocity vector (Utip1)
quiver(Cz2,-CTheta2,0,Utip1,0)
hold on
% Plot Downstream Relative Velocity vector (W2)
quiver (0,0,Wz2,WTheta2,0)

title('Combined Velocity Triangle');
xlabel('m/s');
ylabel('m/s');

% ***********************
% Generate Meriodonal Plot
% ***********************

figure (4)

% Plot outer casing wall profile
quiver(-2*AABC,Params.OCD/2,5*AABC,0,0,'.')
axis([-0.1 0.1 0 0.1])
hold on

% ***********************************************
% Implement blade row ramp spline
% ***********************************************

% Calculate Ramp Slope (RS)
RS = (Params.HER-HIR)/AABC;
disp(['Blade Row Ramp Slope (RS) = ', num2str(RS)])

hub_axial = [0.0 AABC];
hub_radial = [HIR Params.HER];

% Implement blade row ramp spline
CS1 = spline(hub_axial,[RS,hub_radial,RS]);
X1 = linspace(0,AABC,50);
Y1 = fnval(CS1,X1);

% Plot hub profile
plot(X1,Y1,'-g')
hold on

% ***********************************************
% Implement spline for outlet transition
% ***********************************************

transition_axial = [AABC+0.0001 AABC+Params.OT_axial];
```

```
transition_radial = [Params.HER Params.HER+Params.OT_radial];

% Implement outlet transition spline
CS2 = spline(transition_axial,[RS,transition_radial,0]);
X1a = linspace(AABC+0.0001, AABC+Params.OT_axial,50);
Y1a = fnval(CS2,X1a);

% Plot outlet transition profile
plot(X1a,Y1a,'-g')
hold on

% ************************************************
% Implement spinner (Conical)
% ************************************************

Y_delta = Y1(1);
X_delta = Y_delta/RS;

X2 = linspace(0-X_delta,-0.0001,50);
Y2 = (RS*X2)+Y1(1);

plot (X2,Y2, '-r')
hold on

% ************************************************
% Implement axial downstream outlet
% ************************************************

X3 = linspace(AABC+Params.OT_axial+ 0.0001,3*AABC,100);
Y3 = Params.HER+Params.OT_radial*ones(1,100);

plot (X3, Y3, '-b')
hold on

title('Meriodonal Plot');
xlabel('m');
ylabel('m');

% Check to see if the specified upstream and downstream axial wedge lengths
% are actually inside the region of the hub.  Query the user if they wish
% to continue anyway.
if (-Params.UpstreamAxial*Params.OCD > X2(1))   % Is the UpstreamAxial
greater than the first point on the rotor?
    disp('**WARNING**: Params.UpstreamAxial specifies a point that');
    disp('is not past the beginning of the rotor hub.')
    reply = input('Would you like to continue anyway? y/n  ','s');
    validResponse = false;
    while ~validResponse

        switch lower(reply)
            case 'y'
                validResponse = true;
            case 'n'
```

```
                validResponse = true;
                err = MException('MATLAB:InvalidEndpoint', 'Invalid
Params.UpstreamAxial value');
                throw(err)
            otherwise
                disp('Not a valid option.')
                reply = input('Would you like to continue? y/n ','s');
        end
    end
end

if ((Params.DownstreamAxial*Params.OCD + AABC) < X3(end))    % Is the
UpstreamAxial greater than the first point on the rotor?
    disp('**WARNING**: Params.DownstreamAxial specifies a point that');
    disp('is not past the end of the rotor hub.')
    reply = input('Would you like to continue anyway? y/n   ','s');
    validResponse = false;
    while ~validResponse

        switch lower(reply)
            case 'y'
                validResponse = true;
            case 'n'
                validResponse = true;
                err = MException('MATLAB:InvalidEndpoint', 'Invalid
Params.DownstreamAxial value');
                throw(err)
            otherwise
                disp('Not a valid option.')
                reply = input('Would you like to continue? y/n ','s');
        end
    end
end



% Generate points at the front and back of the hub--zeros at the leading
side,
% continuing the extreme point at the trailing side
numPoints = 10;      % Number of points used to generate the linspace before
and after
X0 = linspace(-Params.UpstreamAxial*Params.OCD, X2(1), numPoints)';
% Trim the last point off, to avoid duplicate points
X0 = X0(1:end-1);
Y0 = zeros(size(X0));

X4 = linspace(X3(end), (Params.DownstreamAxial*Params.OCD + AABC),
numPoints)';
% Trim the first point off, to avoid duplicates
X4 = X4(2:end);
Y4 = Y3(end)*ones(size(X4));


X1_final = X1.';
X1a_final = X1a.';
X2_final = X2.';
```

```
X3_final = X3.';
X_hubfinal = [X0; X2_final; X1_final; X1a_final(2:end); X3_final; X4];
```

```
Y1_final = Y1.';
Y1a_final = Y1a.';
Y2_final = Y2.';
Y3_final = Y3.';

Y_hubfinal = [Y0; Y2_final; Y1_final; Y1a_final(2:end); Y3_final; Y4];

% X_casingfinal = X_hubfinal;
% Calculate the casing upstream and downstream length based on the supplied
number of Outer Casing Diameters
X_casingfinal = linspace(-Params.UpstreamAxial*Params.OCD,
Params.DownstreamAxial*Params.OCD + AABC, 250)';


%Y4 = (Params.OCD/2)*ones(1,250);
%Y_casingfinal = Y4.';
Y_casingfinal = (Params.OCD/2)*ones(size(X_casingfinal));

%**************************************************
% Plot X & Y hub & casing coordinates before export
%***********************************************
figure (5)
plot (X_hubfinal, Y_hubfinal, '-r')
hold on
plot (X_casingfinal, Y_casingfinal, '-b')
hold on

axis([-0.15 0.15 0 0.15])

title('X & Y hub & casing coordinate plot for export');
xlabel('m');
ylabel('m');
```

XV. **PASSAGE.M**

```
% m-function file to assign the values of the Sanger passage

function [z,r, Hub] = Passage(N_z,N_r,Blade,Hub,Casing)

% Hub in metric
% Old hard-coded hub coordinates - Commented out in favor of Hub.x and Hub.y
input
if 0
HUB_M = [-0.25398984    0
-0.2033016   0
-0.17778984 0
-0.1524 0
-0.13968984 0
-0.12954    0
-0.124458984    0
-0.11939016 0
-0.10158984 0
-0.1    0
-0.066126614    0
-0.065509394    0.0065786
-0.063652654    0.0131572
-0.060558934    0.0197358
-0.056230774    0.0263144
-0.050663094    0.032893
-0.043855894    0.0394716
-0.035814254    0.0460502
-0.026535634    0.0526288
-0.016020034    0.0592074
-0.004264914    0.065786
0   0.068072
0.025389841 0.08171688
0.03048 0.0844296
0.03557016   0.0871728
0.0381   0.08851392
0.0405384    0.089406984
0.043181016 0.090093698
0.04572 0.090728902
0.04824984   0.091235784
0.050797968 0.091540584
0.05334 0.091945968
0.0557784    0.092202
0.058417968 0.092455898
0.0633984    0.092709797
0.0762   0.093294098
0.1057656    0.093294098
0.1335024    0.090424
0.1524   0.090424
0.2033016    0.090424
0.253998984 0.090424];
end

% Old hard-coded casing coordinates - Commented out in favor of Casing.x and
```

```
Casing.y input
```

```
% Casing in metric
if 0
CASING_M = [-0.25398984 0.13968984
-0.23277576 0.13968984
-0.203199797     0.13968984
-0.1524 0.13968984
-0.101598984     0.13968984
-0.06348984 0.13968984
-0.03776472 0.13968984
-0.012701015     0.13968984
0.01612392   0.13968984
0.06605016   0.13968984
0.09396984   0.13968984
0.1524   0.13968984
0.203197968 0.13968984
0.253998984 0.13968984];
end

HUB_M = [Hub.x Hub.y];
CASING_M = [Casing.x Casing.y];

% Conversion to inches
HUB_M = HUB_M./0.0254; CASING_M = CASING_M./0.0254;

figure(1);close;figure(1);
plot(HUB_M(:,1),HUB_M(:,2),'-r'); axis equal; hold on
plot3(CASING_M(:,1),CASING_M(:,2),zeros(size(CASING_M(:,2))),'-r');
xlabel('z [in]'); ylabel('r [in]');

% Blade leading and trailing edge locations
TempLead  = zeros(size(Blade.z)); % Memory allocation
TempTrail = zeros(size(Blade.z)); % Memory allocation
% Z points have to be shifted to match the blades.
for ii = 1:size(Blade.z,3) % Loop to find forward most points on blade
    TempLead(:,:,ii)   = ones(size(Blade.z,1),1)*min(min(Blade.z,[],1),[],3);
    TempTrail(:,:,ii) = ones(size(Blade.z,1),1)*max(max(Blade.z,[],1),[],3);
end

% Place TempLead and TempTrail values into an array with the actual points,
to avoid issues with ordering
TempLeadPoints.r_unsorted = Blade.r(Blade.z==TempLead);
TempLeadPoints.z_unsorted = Blade.z(Blade.z==TempLead);
TempTrailPoints.r_unsorted = Blade.r(Blade.z==TempTrail);
TempTrailPoints.z_unsorted = Blade.z(Blade.z==TempTrail);
% Sort in order of ascending 'r' values
[TempLeadPoints.r TempLeadPoints.sortIndices] =
sort(TempLeadPoints.r_unsorted);
[TempTrailPoints.r TempTrailPoints.sortIndices] =
sort(TempTrailPoints.r_unsorted);
% Sort the z coordinates to match
TempLeadPoints.z = zeros(size(TempLeadPoints.z_unsorted));
TempTrailPoints.z = zeros(size(TempTrailPoints.z_unsorted));
for ii = 1:length(TempLeadPoints.z)
    TempLeadPoints.z(ii) =
TempLeadPoints.z_unsorted(find(TempLeadPoints.sortIndices == ii));
end
```

```
for ii = 1:length(TempTrailPoints.z)
    TempTrailPoints.z(ii) =
TempTrailPoints.z_unsorted(find(TempTrailPoints.sortIndices == ii));
end

% Axial coordinates
z = ones(N_r,1)*[linspace(CASING_M(1,1),mean(TempLeadPoints.z),N_z(1))
linspace(mean(TempTrailPoints.z),CASING_M(end,1),N_z(2))];
%z = ones(N_r,1)*[CASING_M(1,1):(CASING_M(end,1)-CASING_M(1,1))/(sum(N_z)-
1):CASING_M(end,1)];
r = zeros(size(z));

% Hub radial coordinates
%r(1,:) = interp1(HUB_M(:,1),HUB_M(:,2),z(1,:),'spline','extrap');
r(1,:) = interp1(HUB_M(:,1),HUB_M(:,2),z(1,:));
%r(end,:) = interp1(CASING_M(:,1),CASING_M(:,2),z(end,:),'spline','extrap');
r(end,:) = interp1(CASING_M(:,1),CASING_M(:,2),z(end,:));

% Initial guess of internal radial points
r = sqrt(ones(N_r,1)*r(1,:).^2 + (((0:(N_r-1))'/(N_r-1)))*(r(end,:).^2-
r(1,:).^2));

disp('Blade leading edge')
plot(TempLeadPoints.z,TempLeadPoints.r,'-b')

disp('Blade Trailing edge')
plot(TempTrailPoints.z,TempTrailPoints.r,'-b')

% Loop to find intersections and modify z (and r coords for first run)
for ii = 1:size(r,1)
    % Modification of upstream z coords
    [zi,ri] = polyxpoly(TempLeadPoints.z,TempLeadPoints.r,z(ii,:),r(ii,:));
    plot(zi,ri,'or')
    z(ii,1:N_z(1)) = linspace(z(ii,1),zi,N_z(1));

    % Modification of downstream z coords
    [zi,ri] = polyxpoly(TempTrailPoints.z,TempTrailPoints.r,z(ii,:),r(ii,:));
    plot(zi,ri,'or')
    z(ii,N_z(1)+1:end) = linspace(zi,z(ii,end),N_z(2));
end


% recalc of radial coords
r(1,:) = interp1(HUB_M(:,1),HUB_M(:,2),z(1,:));
r(end,:) = interp1(CASING_M(:,1),CASING_M(:,2),z(end,:));

% Initial guess of internal radial points
r = sqrt(ones(N_r,1)*r(1,:).^2 + (((0:(N_r-1))'/(N_r-1)))*(r(end,:).^2-
r(1,:).^2));

plot(z',r','-+r'); axis equal;

Hub.z = HUB_M(:,1)';
Hub.r = HUB_M(:,2)';
```

XVI. **WEDGEGEN.M**

```
function [Wedge] = WedgeGen(Blade,z,r)
% WedgeGen Function to output the gas wedge geometry and ensure that it
% contains the blades of the splittered rotor
% Wedges
% 1 Inlet
% 1 Outlet
% 1 Blade inlet
% 1 Blade outlet
% 1 For each Splitter blade

% Wedgeoffset is zero if not specified
if ~isfield(Blade, 'WedgeOffset')
    Blade.WedgeOffset = 0;
end

if ~isfield(Blade, 'WedgeNo')
    Blade.WedgeNo = 8;
end

% Range to search within for the wedge
% Make this relative to the Blade.LE
%Wedge.Range = 0.31; % Specifies the range as fraction of blade axial chord
to search for points to construct the wedge
AngleWedge   = (pi/180)*(0.5*360/Blade.PassNo); % Half angle span of wedge in
radians
OffsetWedge = Blade.WedgeOffset*(pi/180)*(360/Blade.PassNo); % Manual offset
temp.TransNo = 2; % Number of transition points

% Memory allocations and basic geometry
Wedge.y = zeros(Blade.S+1,Blade.WedgeNo+2*temp.TransNo+6,2); % 6 extra for
ends and guide curves
Wedge.x = Wedge.y;
Wedge.z = Wedge.y;

Wedge.r   = Wedge.y;
Wedge.th = Wedge.y;

for ii = 1:Blade.S % Each blade station
    temp.Z   = reshape(Blade.z(:,ii,:),[],1); % Reshaped to single column
    temp.X   = reshape(Blade.x(:,ii,:),[],1); % Reshaped to single column
    temp.Y   = reshape(Blade.y(:,ii,:),[],1); % Reshaped to single column
    temp.R   = reshape(Blade.r(:,ii,:),[],1); % Reshaped to single column
    temp.TH = reshape(Blade.th(:,ii,:),[],1); % Reshaped to single column
    %temp.z = linspace(min(temp.Z),max(temp.Z),Blade.WedgeNo);
    temp.z =
linspace(min(min(min(Blade.z))),max(max(max(Blade.z))),Blade.WedgeNo);

    % Interpolate theta points at constant radius
    % Calculate which radius to use for interpolation
    % If this is the first slice, use the maximum radius
    if ii == 1
```

```
    radius = max(max(Blade.r(:,ii,:)));
% If this is the last slice, use the minimum radius
```

```
    elseif ii == Blade.S
        radius = min(min(Blade.r(:,ii,:)));
    % In all other cases, use the mean radius
    else
        radius = max(max(Blade.r(:,1,:)))+(ii-1)/(Blade.S-
1)*(min(min(Blade.r(:,Blade.S,:)))-max(max(Blade.r(:,1,:))));
        %radius = mean(reshape(Blade.r(:,ii,:),[],1));
    end

    figure(2)
    % Interpolate z and theta values at 'radius'
    % Preallocate an array to store these values in
    interpolated.z = ones(size(Blade.r,1),size(Blade.r,3));
    interpolated.th = interpolated.z;
    % Loop around the vertical lines and the blades
    for kk = 1:size(Blade.r,3)
        for jj = 1:size(Blade.r,1)
            interpolated.z(jj,kk) = interp1(Blade.r(jj,:,kk),
Blade.z(jj,:,kk), radius);
            interpolated.th(jj,kk) = interp1(Blade.r(jj,:,kk),
Blade.th(jj,:,kk), radius);
            %[Blade.r(jj,:,kk)' Blade.z(jj,:,kk)' Blade.th(jj,:,kk)']
            [graphy, graphx] = pol2cart(Blade.th(jj,:,kk),Blade.r(jj,:,kk));
            plot3(Blade.z(jj,:,kk),graphy,graphx,'ob')
            %plot3(Blade.z(jj,:,kk),Blade.y(jj,:,kk),Blade.x(jj,:,kk),'ob')
        end
    end


    % Section no longer used because calculations are done with constant r-
values
    % r wedge values are found
    %[P,S,MU] =
polyfit(reshape(Blade.z(:,ii,:),[],1),reshape(Blade.r(:,ii,:),[],1),Blade.Wed
geNo-1);
    %[P,S,MU] =
polyfit(reshape(Blade.z(:,ii,:),[],1),reshape(Blade.r(:,ii,:),[],1),3);
    %temp.r    = polyval(P,temp.z,S,MU);
    temp.r = radius*ones(size(temp.z));

    [interpolated.y, interpolated.x] = pol2cart(interpolated.th,
radius*ones(size(interpolated.th)));
    plot3(interpolated.z, interpolated.y, interpolated.x, 'or')

    % Generate the fit equation
    %[P,S,MU] =
polyfit(reshape(Blade.z(:,ii,:),[],1),reshape(Blade.th(:,ii,:),[],1),Blade.We
dgeNo-1);
    %[P,S,MU] =
polyfit(reshape(interpolated.z,[],1),reshape(interpolated.th,[],1),1);
    [P,S,MU] =
polyfit(reshape(interpolated.z,[],1),reshape(radius*interpolated.th,[],1),1);
    temp.th    = polyval(P,temp.z,S,MU)/radius - OffsetWedge;

    % Graph the fit equation
    %fit.z = (temp.z(1)-temp.TransNo*diff(temp.z(1:2))-
.5:.01:(temp.z(end)+temp.TransNo*diff(temp.z((end-1):end)))+.5;
```

```
    %fit.z =
interp1(r(:,1),z(:,1),temp.r(1),'linear','extrap'):0.01:interp1(r(:,end),z(:,
end),temp.r(end),'linear','extrap');
    %fit.th = polyval(P,fit.z,S,MU);
    %fit.r = radius*ones(size(fit.th));
    %[fit.y fit.x] = pol2cart(fit.th,fit.r);
    %plot3(fit.z,fit.y,fit.x,'r')

    % Function to find the point as far away from each section as possible
    disp(['Find distance to blade section ' ii])
    temp = MinDist(interpolated,temp,radius,AngleWedge);

    figure(2)
    fit.z  = temp.z;
    fit.th = temp.th;
    fit.r = radius*ones(size(fit.th));
    [fit.y, fit.x] = pol2cart(fit.th,fit.r);
    plot3(fit.z,fit.y,fit.x,'r')

    % Additional inlet and outlet points added to ensure smooth transition
    % to the straight inlet and outlet regions
    temp.zi   = linspace(temp.z(1)-temp.TransNo*diff(temp.z(1:2)),temp.z(1)-
diff(temp.z(1:2)),temp.TransNo);
    temp.zo   = linspace(temp.z(end)+diff(temp.z((end-
1):end)),temp.z(end)+temp.TransNo*diff(temp.z((end-1):end)),temp.TransNo);

    % Angle is slowly rotated to be straight
    %fprintf('Inlet Derivative = %f\n', diff(temp.th(1:2))/diff(temp.z(1:2)))
    pp        = spapi(3,[temp.zi(1); temp.z(1)],[0;
diff(temp.th(1:2))/diff(temp.z(1:2))]); % Spline fit of slopes
    temp.thi = (fnval(fnint(pp),[temp.zi temp.z(1)]));
% Integration
    temp.thi = temp.th(1)-fliplr(cumsum(fliplr(diff(temp.thi))));
% Cumulative difference

    %fprintf('Outlet Derivative = %f\n', diff(temp.th(end-
1:end))/diff(temp.z(end-1:end)))
    pp        = spapi(3,[temp.z(end); temp.zo(end)],[diff(temp.th(end-
1:end))/diff(temp.z(end-1:end)); 0]); % Spline fit of slopes
    temp.tho = (fnval(fnint(pp),[temp.z(end) temp.zo]));
% Integration
    temp.tho = temp.th(end)+temp.tho(2:end);

    temp.z    = [temp.zi temp.z temp.zo];
    temp.th = [temp.thi temp.th temp.tho];
    temp.r   = [ones(1,temp.TransNo)*temp.r(1) temp.r
ones(1,temp.TransNo)*temp.r(end)];

    % Inlet and outlet guide curve to keep final sections straight
    temp.z    = [temp.z(1)-diff(temp.z(1:2)) temp.z
temp.z(end)+diff(temp.z(end-1:end))];
    temp.r    = [temp.r(1) temp.r temp.r(end)];
    temp.th   = [temp.th(1) temp.th temp.th(end)];

    % Final inlet and outlet points added
```

```
    temp.z   = [interp1(r(:,1),z(:,1),temp.r(1),'linear','extrap') temp.z
interp1(r(:,end),z(:,end),temp.r(end),'linear','extrap')];
    temp.r   = [temp.r(1) temp.r temp.r(end)];
    temp.th = [temp.th(1) temp.th temp.th(end)];

    % Additional mid lines added to ensure straight long inlets and outlets
    temp.z   = [temp.z(1) mean(temp.z(1:2)) temp.z(2:(end-1)) mean(temp.z(end-
1:end)) temp.z(end)];
    temp.r   = [temp.r(1) temp.r temp.r(end)];
    temp.th = [temp.th(1) temp.th temp.th(end)];

    figure(2)
    [temp.y,temp.x] = pol2cart(temp.th+AngleWedge,temp.r);
    Wedge.y(ii+1,:,1)   = temp.y;
    Wedge.x(ii+1,:,1)   = temp.x;
    Wedge.z(ii+1,:,1)   = temp.z;
    Wedge.r(ii+1,:,1)   = temp.r;
    Wedge.th(ii+1,:,1) = temp.th+AngleWedge;

    [temp.y,temp.x] = pol2cart(temp.th-AngleWedge,temp.r);
    Wedge.y(ii+1,:,2)   = temp.y;
    Wedge.x(ii+1,:,2)   = temp.x;
    Wedge.z(ii+1,:,2)   = temp.z;
    Wedge.r(ii+1,:,2)   = temp.r;
    Wedge.th(ii+2,:,1) = temp.th-AngleWedge;

    plot3(Wedge.z(ii+1,:,1),Wedge.y(ii+1,:,1),Wedge.x(ii+1,:,1),'-og')    %
Start of wedge
    plot3(Wedge.z(ii+1,:,2),Wedge.y(ii+1,:,2),Wedge.x(ii+1,:,2),'-og') % End
of wedge

end % for ii

% Bottom z points of wedge need to be added
Wedge.z(1,:,:) = Wedge.z(2,:,:);

plot3(Wedge.z(1,:,1),Wedge.y(1,:,1),Wedge.x(1,:,1),'-og')
plot3(Wedge.z(:,:,1),Wedge.y(:,:,1),Wedge.x(:,:,1),'-og')
plot3(Wedge.z(:,:,2),Wedge.y(:,:,2),Wedge.x(:,:,2),'-og')

end % function
```

XVII. STREAMGEN.M

```
function [ Stream, tol,dpsi_dr ] = StreamGen( Stream,Air,Inlet )
%StreamGen Calculates streamline radial positions

[dpsi_dz,dpsi_dr]       = d_d(Stream.z,Stream.r,Stream.psi);       %
Differentiation in z,r coords of dpsi/dz & dpsi/dr

% N,S,E,W points are found on an underlying grid
[rN,rS,rE,rW,rn,rs,re,rw]                      =
NEWS(Stream.z,Stream.psi,Stream.r);     % Underlying r, on psi-z grid
[zN,zS,zE,zW,zn,zs,ze,zw]                      =
NEWS(Stream.z,Stream.psi,Stream.z);     % Underlying z, on psi-z grid
[psiN,psiS,psiE,psiW,psin,psis,psie,psiw] =
NEWS(Stream.z,Stream.psi,Stream.psi); % Underlying psi, on psi-z grid

% Grid spaces used
dzw = Stream.z-zW; dze = zE-Stream.z; dz =0.5*(dzw+dze);
dpsi = psin-psis; dpsin = psiN-Stream.psi; dpsis = Stream.psi-psiS;

[dr_dz,dr_dpsi]         = d_d(Stream.z,Stream.psi,Stream.r);             %
Differentiation in z,psi coords of dr/dz & dr/dpsi
[drcth_dz,drcth_dpsi]   = d_d(Stream.z,Stream.psi,Stream.r.*Stream.Cth); %
Differentiation in z,psi coords of d(r Cth)/dz & d(r Cth)/dpsi
[dr2_dzdpsi,dr2_d2psi] = d_d(Stream.z,Stream.psi,dr_dpsi);               %
Differentiation in z,psi coords of d2r/(dz dpsi) & d2r/(dpsi2)
[dho_dz,dho_dpsi]       = d_d(Stream.z,Stream.psi,Stream.ho);
% Differentiation in z,psi coords of dho/dz & dho/dpsi
[ds_dz,ds_dpsi]         = d_d(Stream.z,Stream.psi,Stream.s);             %
Differentiation in z,psi coords of ds/dz & ds/dpsi

% For centerline points the slope of Velocity is zero
if 0
    if not(isempty(find(r(1,:)==0)))
        temp = (find(r(1,:)==0));
        dr_dpsi(1,temp) = dr_dpsi(2,temp) - (r(2,temp).^2).*(dr_dpsi(3,temp)-
dr_dpsi(2,temp))./(r(3,temp).^2 - r(2,temp).^2);
    end
end

% Backflow prevention
if not(isempty(find(dpsi_dr(1,:)<0)))
    disp('Backflow present')
    dpsi_dr(1,find(dpsi_dr(1,:)<0)) = zeros(size(find(dpsi_dr(1,:)<0)));
end

% Axial Velocities
Cz       = (1./(Stream.rho.*Stream.r)).*dpsi_dr; % Axial velocity
Cz       = extrap(Stream.r,Cz);
% Radial Velocities
Cr       = Cz.*dr_dz;               % Radial velocity
if not(isempty(find(Stream.r(1,:)==0))) % Centreline radial velocity is zero
    disp('Centre present')
```

```
temp        = (find(Stream.r(1,:)==0));
```

```
    Cz(1,temp) = Cz(2,temp) - (Stream.r(2,temp).^2).*(Cz(3,temp)-
Cz(2,temp))./(Stream.r(3,temp).^2 - Stream.r(2,temp).^2);
    Cr(1,temp) = 0;
end

% Gas properties at each grid point
C           = sqrt(Cz.^2 + Cr.^2 + Stream.Cth.^2);          % Resultant
velocity
X           = C./sqrt(2*Stream.ho);                          % Non-dimensional
velocity
Stream.rho = Stream.rho_o.*((1-X.^2).^(1/(Air.Gam-1)));     % Static density
p           = Stream.po.*((1-X.^2).^(Air.Gam/(Air.Gam-1))); % Static pressure
h           = Stream.ho.*(1-X.^2);                           % Static enthalpy
Stream.s    = Air.Cp*log(h/Inlet.ho)-Air.R*log(p/Inlet.Po); % Entropy

% North, south, east, west points of density
[rhoN,rhoS,rhoE,rhoW,rhon,rhos,rhoe,rhow] =
NEWS(Stream.z,Stream.psi,Stream.rho);

% A very simple iterative scheme is used ref from Gannon 1997.
term1 = (Stream.rho.*Stream.r.*(dr_dpsi.^2)./dz).*(rE./(rhoe.*re.*dze) +
rW./(rhow.*rw.*dzw));
term2 =
((1+dr_dz.^2)./(Stream.rho.*Stream.r.*dpsi)).*((rhon.*rn.*rN./dpsin)+(rhos.*r
s.*rS./dpsis));
term3 = ((Stream.rho.*Stream.r).^2).*(dr_dpsi.^3).*(dho_dpsi-
(h./Air.Cp).*ds_dpsi-(Stream.Cth./Stream.r).*drcth_dpsi); % Source term
term4 = 2.*dr_dpsi.*dr_dz.*dr2_dzdpsi;
numer = term1 + term2 + term3 - term4;   % Numerator

term5 = (Stream.rho.*Stream.r.*(dr_dpsi.^2)./dz   ).*(1./(rhoe.*re.*dze) +
1./(rhow.*rw.*dzw));
term6 = ((1+dr_dz.^2)./(Stream.rho.*Stream.r.*dpsi)).*(rhon.*rn./dpsin +
rhos.*rs./dpsis);
denom = term5+term6;

r_old = Stream.r;
Stream.r(2:end-1,2:end-1) = numer(2:end-1,2:end-1)./denom(2:end-1,2:end-1);
% Constant slope inlet region
Stream.r(2:end-1,1)         = Stream.r(2:end-1,2) - diff(Stream.z(2:end-
1,1:2),[],2).*...
    diff(Stream.r(2:end-1,2:3),[],2)./diff(Stream.z(2:end-1,2:3),[],2);
% Constant slope outlet region
Stream.r(2:end-1,end)       = Stream.r(2:end-1,end-1) + ...
    diff(Stream.z(2:end-1,end-1:end),[],2).*...
    diff(Stream.r(2:end-1,end-2:end-1),[],2)./...
    diff(Stream.z(2:end-1,end-2:end-1),[],2);

Stream.r(2:end-1,2:end-1) = Stream.relax*Stream.r(2:end-1,2:end-1) + (1-
Stream.relax)*r_old(2:end-1,2:end-1);

tol = max(max(abs((Stream.r(2:end-1,:)-r_old(2:end-1,:))./r_old(2:end-
1,:))));

% Velocities are assigned
```

```
Stream.Cz = Cz; % Axial
Stream.Cr = Cr; % Radial
Stream.C   = C;  % Absolute

end
```

XVIII. D_D.M

```
% m-function file to find the first derivative of two variables
% First two inputs, z, psi are underlying grid
% Third input is the variable to be differentiated wrt to the grid

function [dr_dz,dr_dpsi] = d_d(z,psi,r)

% Memory allocation
dr_dz   = zeros(size(z));
dr_dpsi = dr_dz;

PG1 = dr_dz; PG2 = dr_dz; ZG1 = dr_dz; ZG2 = dr_dz;

% Local rectangular grid
% Problem lies here
[rN,rS,rE,rW,rn,rs,re,rw]                         = NEWS(z,psi,r);
[zN,zS,zE,zW,zn,zs,ze,zw]                         = NEWS(z,psi,z);
[psiN,psiS,psiE,psiW,psin,psis,psie,psiw] = NEWS(z,psi,psi);
% Method of Greyvenstein from Gannon's masters (pg45) is used

% Differentiation in the psi direction
d_psi = psin-psis; d_psi_s = psi-psiS; d_psi_n = psiN-psi;
PG1(2:end-1,:) = d_psi_s(2:end-1,:)./(2*d_psi(2:end-1,:).*d_psi_n(2:end-
1,:));
PG2(2:end-1,:) = d_psi_n(2:end-1,:)./(2*d_psi(2:end-1,:).*d_psi_s(2:end-
1,:));

dr_dpsi = PG1.*(rN-r) + PG2.*(r-rS);

% Differentiation in the z direction
%d_z = ze-zw; d_z_w = z-zW; d_z_e = zE-z;
d_z_w = z-zW; d_z_e = zE-z; d_z = 0.5*(d_z_w + d_z_e);
ZG1(:,2:end-1) = d_z_w(:,2:end-1)./(2*d_z(:,2:end-1).*d_z_e(:,2:end-1));
ZG2(:,2:end-1) = d_z_e(:,2:end-1)./(2*d_z(:,2:end-1).*d_z_w(:,2:end-1));

dr_dz = ZG1.*(rE-r) + ZG2.*(r-rW);

% Bottom points, simple linear extrapolation to sides used
%dr_dpsi(1,:)    = dr_dpsi(2,:) -
diff(dr_dpsi(2:3,:)).*d_psi_s(2,:)./d_psi_n(2,:);

% Bottom points, quadratic fit, somewhat compuationally expensive but well
% worth the programming fun
y2 = dr_dpsi(2,:); y3 = dr_dpsi(3,:); y4 = dr_dpsi(4,:);
x2 =     psi(2,:); x3 =     psi(3,:); x4 =     psi(4,:);

a = -(y4.*x3-y4.*x2-y2.*x3+x4.*y2-x4.*y3+x2.*y3)./(-
x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);
b = (-x2.^2.*y4+x2.^2.*y3+x3.^2.*y4-y2.*x3.^2-y3.*x4.^2+y2.*x4.^2)./(-
x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);
c = (x2.^2.*y4.*x3-x2.^2.*x4.*y3-x3.^2.*y4.*x2+y3.*x4.^2.*x2-
```

```
y2.*x4.^2.*x3+x3.^2.*x4.*y2)./(-x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-
x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);
```

```
dr_dpsi(1,:) = a.*psi(1,:).^2 + b.*psi(1,:) + c;

% Top points, simple linear extrapolation to sides used
dr_dpsi(end,:) = dr_dpsi(end-1,:) +...
    diff(dr_dpsi(end-2:end-1,:)).*d_psi_n(end-1,:)./d_psi_s(end-1,:);

% Inflow, linear extrapolation of slope
%dy_dr(:,1) = dy_dr(:,2) -
diff(dy_dr(:,2:3),[],2).*diff(z(:,1:2),[],2)./diff(z(:,2:3),[],2);

% Inflow, constant slope
dr_dz(:,1) = dr_dz(:,2);

% Outflow, linear extrapolation of slope
%dy_dr(:,end) = dy_dr(:,end-1) + diff(dy_dr(:,end-2:end-
1),[],2).*diff(z(:,end-1:end),[],2)./diff(z(:,end-2:end-1),[],2);

% Outflow, constant slope
dr_dz(:,end) = dr_dz(:,end-1);
```

XIX. **NEWS.M**

```
% m-function file to find the north, south, east and west points from a
% non-uniform grid

function [NORTH,SOUTH,EAST,WEST,north,south,east,west] = NEWS(z,psi,r)

NORTH = ones(size(z));
SOUTH = ones(size(z));
EAST  = ones(size(z));
WEST  = ones(size(z));
north = ones(size(z));
south = ones(size(z));
east  = ones(size(z));
west  = ones(size(z));
%dr     = ones(size(z));
%dre    = ones(size(z));
%drw    = ones(size(z));

% Sets up interpolation method FL (linear)
FL =
TriScatteredInterp(reshape(z,(numel(z)),1),reshape(psi,(numel(psi)),1),reshap
e(r,(numel(r)),1),'linear');
% Sets up interpolation method FN (nearest) for exterior points
FN =
TriScatteredInterp(reshape(z,(numel(z)),1),reshape(psi,(numel(psi)),1),reshap
e(r,(numel(r)),1),'nearest');

% Major NORTH points
temp.z = z(2:end-1,:); temp.psi = psi(3:end,:);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
NORTH(2:end-1,:) = temp.r;

% Major SOUTH points
temp.z = z(2:end-1,:); temp.psi = psi(1:end-2,:);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Catch for exterior points
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
SOUTH(2:end-1,:) = temp.r;

% Major WEST points
temp.z = z(:,1:end-2); temp.psi = psi(:,2:end-1);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
```

```
WEST(:,2:end-1) = temp.r;
```

```
% Major EAST points
temp.z = z(:,3:end); temp.psi = psi(:,2:end-1);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
EAST(:,2:end-1) = temp.r;

% Minor north points
temp.z = z(2:end-1,:); temp.psi = 0.5*(psi(2:end-1,:)+psi(3:end,:));
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
north(2:end-1,:) = temp.r;

% Minor south points
temp.z = z(2:end-1,:); temp.psi = 0.5*(psi(1:end-2,:)+psi(2:end-1,:));
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
south(2:end-1,:) = temp.r;

% Minor west points
temp.z = 0.5*(z(:,1:end-2)+z(:,2:end-1)); temp.psi = psi(:,2:end-1);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
west(:,2:end-1) = temp.r;

% Minor east points
temp.z = 0.5*(z(:,2:end-1)+z(:,3:end)); temp.psi = psi(:,2:end-1);
temp.r = FL(temp.z,temp.psi);
if max(reshape(isnan(temp.r),numel(temp.r),1)) % Exterior point catch
    temp.r(isnan(temp.r)) =
FN(temp.z(isnan(temp.r)),temp.psi(isnan(temp.r)));
end
east(:,2:end-1) = temp.r;
```

XX. EXTRAP.M

```
% M-file to perform quadratic extrapolation to the bottom of a matrix. It
% uses the first three internal points to back out the edge point by
% fitting a quadratic polynomial to the data

function [y] = extrap(x,y)

y2 = y(2,:); y3 = y(3,:); y4 = y(4,:);
x2 = x(2,:); x3 = x(3,:); x4 = x(4,:);

a = -(y4.*x3-y4.*x2-y2.*x3+x4.*y2-x4.*y3+x2.*y3)./(-
x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);
b = (-x2.^2.*y4+x2.^2.*y3+x3.^2.*y4-y2.*x3.^2-y3.*x4.^2+y2.*x4.^2)./(-
x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);
c = (x2.^2.*y4.*x3-x2.^2.*x4.*y3-x3.^2.*y4.*x2+y3.*x4.^2.*x2-
y2.*x4.^2.*x3+x3.^2.*x4.*y2)./(-x4.^2.*x3+x4.^2.*x2+x2.^2.*x3-
x4.*x2.^2+x4.*x3.^2-x2.*x3.^2);

y(1,:) = a.*x(1,:).^2 + b.*x(1,:) + c;

if 0
for i = 1:size(x,2)
    figure(1); close; figure(2);
    plot(x(1:4,i),y(1:4,i))
    pause
end
end
```

XXI. SOLIDWORKSGEN.M

```
% Script to draw the SolidWorks wedge

% SolidWorks drawing generation
h1     = NET.addAssembly('C:\Program Files\SolidWorks
Corp\SolidWorks\SolidWorks.Interop.sldworks.dll'); % 2010 version
%h1     = NET.addAssembly('C:\Program Files\SolidWorks Corp\SolidWorks
(2)\SolidWorks.Interop.sldworks.dll') % Newer version
swApp = SolidWorks.Interop.sldworks.SldWorksClass;

% Make application visible
if ~(swApp.Visible)
    swApp.Visible = true;
end

Part   = swApp.OpenDoc6([pwd '\BasicWedge.SLDPRT'], 1, 0, [], 0,0);

% This allows geometries of less than 1mm
Part.SketchManager.AddToDB = true;
Part.SketchManager.DisplayWhenAdded = false;

%------------------------------------------------------------------------
% Guide Curves for computational volume  are created
count.Curve           = 1;
count.WedgeGuideFront = 1;
count.WedgeGuideRear  = 1;
for ii = 1:size(Wedge.z,2)
    % Front side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length(Wedge.z(:,ii,1))
        boolstatus =
Part.InsertCurveFilePoint(Wedge.z(jj,ii,1),Wedge.y(jj,ii,1),Wedge.x(jj,ii,1))
;
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['FrontWedgeGuideCurve' num2str(count.WedgeGuideFront)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve       = count.Curve + 1;
    count.WedgeGuideFront = count.WedgeGuideFront +1;

    % Back side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length(Wedge.z(:,ii,1))
        boolstatus =
Part.InsertCurveFilePoint(Wedge.z(jj,ii,2),Wedge.y(jj,ii,2),Wedge.x(jj,ii,2))
;
```

```
end
boolstatus = Part.InsertCurveFileEnd;
```

```
    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['RearWedgeGuideCurve' num2str(count.WedgeGuideRear)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.WedgeGuideRear = count.WedgeGuideRear +1;
end % for ii
disp('Wedge guide curves generated')

% Streamwise Curves for computational volume are generated
count.WedgeStreamFront = 1;
count.WedgeStreamRear   = 1;
for ii = 1:size(Wedge.z,1)
    % Front side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(ii,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(ii,jj,1)),
(Wedge.y(ii,jj,1)), (Wedge.x(ii,jj,1)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['FrontWedgeStreamCurve' num2str(count.WedgeStreamFront)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve       = count.Curve + 1;
    count.WedgeStreamFront = count.WedgeStreamFront +1;

    % Back side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(ii,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(ii,jj,2)),
(Wedge.y(ii,jj,2)), (Wedge.x(ii,jj,2)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['RearWedgeStreamCurve' num2str(count.WedgeStreamRear)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.WedgeStreamRear = count.WedgeStreamRear +1;
end
disp('Wedge stream curves generated')

% Front Surface of gas computational volume is created
```

```
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.WedgeStreamFront-1)
    CurveName   = ['FrontWedgeStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.WedgeGuideFront-1)
    CurveName   = ['FrontWedgeGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0)

% Rear Surface of gas computational volume is created
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.WedgeStreamRear-1)
    CurveName   = ['RearWedgeStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.WedgeGuideRear-1)
    CurveName   = ['RearWedgeGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0)

% Top surface for the computational volume is created
count.CaseStream = 1;
for ii = 1:2;  % Stream direction curves are placed
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(end,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(end,jj,ii)),
(Wedge.y(end,jj,ii)), (Wedge.x(end,jj,ii)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['CaseStreamCurve' num2str(count.CaseStream)]; boolstatus
    = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
```

```
    count.Curve       = count.Curve + 1;
    count.CaseStream = count.CaseStream + 1;
end


count.CaseGuide = 1;
for ii = 1:length(Wedge.z(end,:,1)) % Guide direction curves are placed
    Part.InsertCurveFileBegin
    % First point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,1),
Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    % Intermediate point
    [TH(1),R(1)] = cart2pol(Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    [TH(2),R(2)] = cart2pol(Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    [temp.Y temp.X temp.Z] = pol2cart(mean(TH), mean(R),
mean([Wedge.z(end,ii,1) Wedge.z(end,ii,2)]));
    boolstatus = Part.InsertCurveFilePoint(temp.Z, temp.Y, temp.X);
    % Last point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,2),
Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['CaseGuideCurve' num2str(count.CaseGuide)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.CaseGuide = count.CaseGuide + 1;

end

% Casing surface of computational volume is created
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.CaseStream-1)
    CurveName   = ['CaseStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.CaseGuide-1)
    CurveName   = ['CaseGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0)

% Front bound surface of computational volume is created
% This is the same as for the sides of the control volume but just the front
and back
```

```
count.BoundEnd          = 1;
for ii = [1 size(Wedge.z,2)]
    for kk = 1:2 % Sides of bound surface are created
        Part.InsertCurveFileBegin
        for jj = 1:length(Wedge.z(:,ii,1))
            boolstatus =
Part.InsertCurveFilePoint(Wedge.z(jj,ii,kk),Wedge.y(jj,ii,kk),Wedge.x(jj,ii,k
k));
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName   = ['Curve' num2str(count.Curve)];
        boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES',
0, 0, 0, false, 0, [], 0);
        CurveName   = ['BoundEndCurve' num2str(count.BoundEnd)];
        boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0,
true, false, CurveName);
        count.Curve     = count.Curve + 1;
        count.BoundEnd = count.BoundEnd +1;
    end % for kk

    % The top curves of the bound surfaces are created
    Part.InsertCurveFileBegin
    % First point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,1),
Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    % Intermediate point
    [TH(1),R(1)] = cart2pol(Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    [TH(2),R(2)] = cart2pol(Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    [temp.Y temp.X temp.Z] = pol2cart(mean(TH), mean(R),
mean([Wedge.z(end,ii,1) Wedge.z(end,ii,2)]));
    boolstatus = Part.InsertCurveFilePoint(temp.Z, temp.Y, temp.X);
    % Last point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,2),
Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['BoundEndCurve' num2str(count.BoundEnd)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.BoundEnd = count.BoundEnd +1;

end % for ii

% Lofted ends are used as filled surface macro not working
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
boolstatus = Part.Extension.SelectByID2('BoundEndCurve1', 'REFERENCECURVES',
0, 0, 0, false, 1, [], 0);
```

```
boolstatus = Part.Extension.SelectByID2('BoundEndCurve2', 'REFERENCECURVES',
0, 0, 0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve3', 'REFERENCECURVES',
0, 0, 0, true, 4098, [], 0);
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0)

Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
boolstatus = Part.Extension.SelectByID2('BoundEndCurve4', 'REFERENCECURVES',
0, 0, 0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve5', 'REFERENCECURVES',
0, 0, 0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve6', 'REFERENCECURVES',
0, 0, 0, true, 4098, [], 0);
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0)
Part.ClearSelection2(true) % Selections are cleared

% Solid is sewn together
boolstatus = Part.Extension.SelectByID2('Surface-Loft1', 'SURFACEBODY', 0, 0,
0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft2', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft3', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft4', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft5', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
myFeature = Part.FeatureManager.InsertSewRefSurface(true, true, true,
0.00011684, 0.0001);
Part.ClearSelection2(true) % Selections are cleared

% Streamlines are generated
count.Streamline = 1;
for ii = 1:N_r
    if ii ~= 1 % General streamlines
        Part.InsertCurveFileBegin
        for jj = 1:length((Stream.z(ii,:)))
            boolstatus = Part.InsertCurveFilePoint(Stream.z(ii,jj),
Stream.r(ii,jj), zeros(size(Stream.z(ii,jj))));
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName         = ['Curve' num2str(count.Curve)];
        boolstatus        = Part.Extension.SelectByID2(CurveName,
'REFERENCECURVES', 0, 0, 0, false, 0, [], 0);
        CurveName         = ['Streamline' num2str(count.Streamline)];
        boolstatus        = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0,
0, true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.Streamline = count.Streamline + 1;

    else % Hub needs to be treated separately, zero points ignored
         % Previously used first row of Stream.r/z, currently uses a distinct
point set
```

```
        Part.InsertCurveFileBegin
        for jj = find([diff(HubProfile.r~=0) 0]==1):length((HubProfile.z))
            boolstatus      = Part.InsertCurveFilePoint(HubProfile.z(jj),
HubProfile.r(jj), zeros(size(HubProfile.z(jj))));
            temp.HubZ(1+jj-find([diff(HubProfile.r~=0) 0]==1))   =
HubProfile.z(jj);
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName         = ['Curve' num2str(count.Curve)];
        boolstatus        = Part.Extension.SelectByID2(CurveName,
'REFERENCECURVES', 0, 0, 0, false, 0, [], 0);
        CurveName         = ['Streamline' num2str(count.Streamline)];
        boolstatus        = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0,
0, true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.Streamline = count.Streamline + 1;

    end
end

Part.ClearSelection2(true)

%-----------------------------------------------------------------------------
% Top surface is trimmed
% Centerline or rotation axis is placed
temp.Z = [min(min(Wedge.z(:,:,1))) max(max(Wedge.z(:,:,1)))];        % Extents
of wedge
temp.Z = [temp.Z(1)-0.1*diff(temp.Z) temp.Z(end)+0.1*diff(temp.Z)]; % 10%
added to beggining and end
temp.Y = [0 0];
temp.X = temp.Y;

% Sketch of rotational axis is placed
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true) % Sketch is opened
skSegment = Part.SketchManager.CreateLine(temp.Z(1), temp.Y(1), temp.X(1),
temp.Z(2), temp.Y(2), temp.X(2));
Part.SketchManager.InsertSketch(true) % Sketch is closed
Part.ClearSelection2(true)

% Top box that is combined with outer streamline it placed
temp.Z = [Stream.z(end,1) temp.Z(1) temp.Z(1) temp.Z(end) temp.Z(end)
Stream.z(end,end)];
R = max(max(max(Blade.r)));
temp.Y = [Stream.r(end,1) Stream.r(end,1) 1.1*R 1.1*R Stream.r(end,end)
Stream.r(end,end)];
temp.X = zeros(size(temp.Y));
clear R

boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true)
for ii = 1:(length(temp.Z)-1)
```

```
    skSegment = Part.SketchManager.CreateLine(temp.Z(ii), temp.Y(ii),
temp.X(ii), temp.Z(ii+1), temp.Y(ii+1), temp.X(ii+1));
end
boolstatus = Part.Extension.SelectByID2('Streamline12', 'REFERENCECURVES', 0,
0, 0, true, 0, [], 0); % Bounding streamline is used
boolstatus = Part.SketchManager.SketchUseEdge2(false);
Part.SketchManager.InsertSketch(true) % Sketch is closed
Part.ClearSelection2(true)

% Revolved cut is performed (this is for SW 2012 version)
%boolstatus = Part.Extension.SelectByID2('Line1@Sketch1', 'EXTSKETCHSEGMENT',
0, 0, 0, false, 16, [], 0)
%boolstatus = Part.Extension.SelectByID2('Sketch2', 'SKETCH', 0, 0, 0, true,
2, [], 0)
%myFeature1 = Part.FeatureManager.FeatureRevolve2(true, true, false, true,
false, false, 0, 0, 6.2831853071796, 0, false, false, 0.00254, 0.00254, 0, 0,
0, true, true, true)
%Part.ClearSelection2(true)

% Revolved cut is performed (this is for SW 2010 version)
boolstatus = Part.Extension.SelectByID2('Line1@Sketch1', 'EXTSKETCHSEGMENT',
0, 0, 0, false, 4, [], 0); % Axis of revolution
boolstatus = Part.Extension.SelectByID2('Sketch2', 'SKETCH', 0, 0, 0, true,
2, [], 0);              % Sketch for cut
myFeature = Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0,
0, true, true); % Revolved cut

Part.ClearSelection2(true)

%--------------------------------------------------------------------------
% Hub cutout is performed
temp.Z = [min(min(Wedge.z(:,:,1))) max(max(Wedge.z(:,:,1)))];        % Extents
of wedge
temp.Z = [temp.HubZ(1) temp.Z(end)+0.1*diff(temp.Z)]; % 10% added end and
streamline beggining used
temp.Z = [temp.Z temp.Z(end) Stream.z(1,end)];   % Rest of sketch is added
temp.Y = [0 0 Stream.r(1,end) Stream.r(1,end)];
temp.X = zeros(size(temp.Z));

boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0); % Plane for sketch
Part.SketchManager.InsertSketch(true) % Sketch is opened

for ii = 1:(length(temp.Z)-1)
    skSegment = Part.SketchManager.CreateLine(temp.Z(ii), temp.Y(ii),
temp.X(ii), temp.Z(ii+1), temp.Y(ii+1), temp.X(ii+1));
end

% Hub streamline is used in sketch
boolstatus = Part.Extension.SelectByID2('Streamline1', 'REFERENCECURVES', 0,
0, 0, true, 0, [], 0);
boolstatus = Part.SketchManager.SketchUseEdge2(false);

Part.SketchManager.InsertSketch(true) % Sketch is closed
Part.ClearSelection2(true)
```

```
% Revolved cut is performed (this is for SW 2010 version)
boolstatus = Part.Extension.SelectByID2('Line1@Sketch1', 'EXTSKETCHSEGMENT',
0, 0, 0, false, 4, [], 0); % Axis of revolution
boolstatus = Part.Extension.SelectByID2('Sketch3', 'SKETCH', 0, 0, 0, true,
2, [], 0);              % Sketch for cut
myFeature = Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0,
0, true, true); % Revolved cut

Part.ClearSelection2(true)

if 0

% Blade elements are generated
for ii = 1:Blade.S
    for jj = 1:(size(Blade.LE,2)-1)
        %Set up constructions and curves by table
        %objcurve   = objPart.Constructions
        %objcurve1 = objcurve.CurvesByTable

        filename = ['BladeSect_' num2str(ii) '_' num2str(jj) '.xls' ];
        filename = [DirectoryName filename];

        objcurve2 = objcurve1.Add(filename)
        set(objcurve2, 'Closure', 'igClosed')        % Closure (igClosed,
igNormal)
        set(objcurve2, 'ClosureType', 'igPeriodic') % Closure Type
(igNatural, igPeriodic)
        set(objcurve2, 'CurveType', 'igDirectFit')   % Controls Curve Fit

    end % for jj
end % for ii

% Attempt at bounded surface in SolidEdge (Finally gave up and went to
SolidWorks which is much simpler)
% objsurf = objcurve.SurfaceByBoundaries % This works
% get(objcurve.CurvesByTables.Item('CurveByTable_1')) % This works to select
% objcurve.CurvesByTables.methods % shows methods associated with an object
% objsurf1 = objsurf.Add(1,objcurve2) % This does not

%disp('paused before kill')
%pause
%! taskkill /F /IM edge.exe /T
end % if 0


%------------------------------------------------------------------------------
% Cut out blades

% Guide Circles for computational volume  are created
% count.Curve             = 1; % Only necessary with isolated following code
count.BladeGuideCircle = 1;
count.BladeNumber       = 1;
for kk = 1:size(Blade.z,3)
```

```
    for ii = 1:size(Blade.z,2)
        % Front side of blade
        Part.InsertCurveFileBegin
        for jj = 1:(length(Blade.z(:,ii,1))+1)
            if jj~=(length(Blade.z(:,ii,1))+1)
                boolstatus =
Part.InsertCurveFilePoint(Blade.z(jj,ii,kk),Blade.y(jj,ii,kk),Blade.x(jj,ii,k
k));
            else
                boolstatus =
Part.InsertCurveFilePoint(Blade.z(1,ii,kk),Blade.y(1,ii,kk),Blade.x(1,ii,kk))
;
            end
        end

        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName   = ['Curve' num2str(count.Curve)];
        boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES',
0, 0, 0, false, 0, [], 0);
        CurveName   = ['Blade' num2str(count.BladeNumber) 'GuideCircle'
num2str(count.BladeGuideCircle)];
        boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0,
true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.BladeGuideCircle = count.BladeGuideCircle +1;
    end % for ii
    count.BladeGuideCircle = 1;
    count.BladeNumber = count.BladeNumber+1;
end % for kk
disp('Blade guide circles generated')
Part.ClearSelection2(true)

% Guide Curves for Computational Volume are Created
count.BladeNumber = 1;
count.BladeGuideCurve = 1;
for kk = 1:size(Blade.z,3)
    for ii = 1:2:size(Blade.x,1)
        Part.InsertCurveFileBegin
        for jj = 1:length(Blade.y(ii,:,:))
            boolstatus =
Part.InsertCurveFilePoint(Blade.z(ii,jj,kk),Blade.y(ii,jj,kk),Blade.x(ii,jj,k
k));
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName   = ['Curve' num2str(count.Curve)];
        boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES',
0, 0, 0, false, 0, [], 0);
        CurveName   = ['Blade' num2str(count.BladeNumber) 'GuideCurve'
num2str(count.BladeGuideCurve)];
        boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0,
true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.BladeGuideCurve = count.BladeGuideCurve +1;
```

```
    end % for ii
    count.BladeGuideCurve = 1;
    count.BladeNumber = count.BladeNumber+1;
end % for kk
disp('Blade guide curves generated')
Part.ClearSelection2(true)

% Generate Cut from Guides - Automated with Loops
% Outermost loop iterates through blades
for jj = 1:size(Blade.z,3)
    % Select the guide circles
    % First time only uses false in the selecting function
    boolstatus = Part.Extension.SelectByID2(['Blade' num2str(jj)
'GuideCircle1'], 'REFERENCECURVES', 0, 0, 0, false, 1, [], 0);
    for ii=2:size(Blade.z, 2)
        boolstatus = Part.Extension.SelectByID2(['Blade' num2str(jj)
'GuideCircle' num2str(ii)], 'REFERENCECURVES', 0, 0, 0, true, 1, [], 0);
    end
    % Select the guide curves
    count.BladeGuideCurve=2; % For this, stores the strange numbers used
inside the function call
    % NOTE: The "/2" below is to take into account the fact that only every
    % other guide curve was drawn from the data. This will need to be changed
    % if a different interval is used.
    for ii=1:(size(Blade.x, 1)/2)
        count.BladeGuideCurve = count.BladeGuideCurve + 4096;
        boolstatus = Part.Extension.SelectByID2(['Blade' num2str(jj)
'GuideCurve' num2str(ii)], 'REFERENCECURVES', 0, 0, 0, true,
count.BladeGuideCurve, [], 0);
    end
    Part.FeatureManager.InsertCutBlend( false, true, false, 1, 0, 0, false,
0, 0, 0, true, true);
end

% Save model as a parasolid
% Only saves the solid geometry, no curves.
longstatus = Part.SaveAs3([pwd '\AirWedge.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\AirWedge.SLDPRT'], 0, 2);
disp('Air wedge without fillets generated. AirWedge.X_T saved.')

% Save air wedge in two pieces, to match rotating sections
% Cutting plane corresponds with end of hub slice
MaxSpace = 0.1*(max(max(max(Blade.z))) - min(min(min(Blade.z)))); % How far
from the fillets the hub should extend on the 'max' side
temp.sliceX = max(max(max(Blade.z))) + Blade.Fillet + MaxSpace;
wedgeHeight = max(((Wedge.y(end,:,1)).^2 + (Wedge.x(end,:,1)).^2).^(1/2));
temp.sliceTop = 1.1*wedgeHeight;
temp.sliceBase = -0.1*wedgeHeight;

boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0); % Plane for sketch
Part.SketchManager.InsertSketch(true); % Sketch is opened
skSegment = Part.SketchManager.CreateLine(temp.sliceX, temp.sliceTop, 0,
temp.sliceX, temp.sliceBase, 0);
Part.SketchManager.InsertSketch(true); % Sketch is closed
```

```
% Generate the surface to cut with
Part.FeatureExtruRefSurface2( false, false, false, 1, 1, 0.00254, 0.00254,
false, false, false, false, 0.01745329251994, 0.01745329251994, false, false,
false, false);

Part.InsertCutSurface( false, 0);

boolstatus = Part.Extension.SelectByID2('Surface-Extrude1', 'REFSURFACE', 0,
0, 0, false, 0, [], 0);
Part.FeatureManager.HideBodies;
Part.ClearSelection2( true);

% Save model as a parasolid
% Only saves the solid geometry, no curves.
longstatus = Part.SaveAs3([pwd '\AirWedge_Upstream.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\AirWedge_Upstream.SLDPRT'], 0, 2);
disp('Air wedge without fillets, upstream half generated.
AirWedge_Upstream.X_T saved.')

% Delete and change direction of surface cut
boolstatus = Part.Extension.SelectByID2('Surface-Extrude1', 'REFSURFACE', 0,
0, 0, false, 0, [], 0);
Part.FeatureManager.ShowBodies;
Part.ClearSelection2( true)
boolstatus = Part.Extension.SelectByID2('SurfaceCut1', 'BODYFEATURE', 0, 0,
0, false, 0, [], 0);
Part.EditDelete;
boolstatus = Part.Extension.SelectByID2('Surface-Extrude1', 'SURFACEBODY', 0,
0, 0, true, 0, [], 0);
Part.InsertCutSurface( true, 0);

boolstatus = Part.Extension.SelectByID2('Surface-Extrude1', 'REFSURFACE', 0,
0, 0, false, 0, [], 0);
Part.FeatureManager.HideBodies;
Part.ClearSelection2( true);

% Save model as a parasolid
% Only saves the solid geometry, no curves.
longstatus = Part.SaveAs3([pwd '\AirWedge_Downstream.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\AirWedge_Downstream.SLDPRT'], 0, 2);
disp('Air wedge without fillets, downstream half generated.
AirWedge_Downstream.X_T saved.')


% This returns the program to normal operation
Part.SketchManager.AddToDB = false;
Part.SketchManager.DisplayWhenAdded = true;
```

XXII. BLADEHUB_WEDGE_CUTOUT.M

```
% Script to draw the rotor shape

% SolidWorks drawing generation
h1     = NET.addAssembly('C:\Program Files\SolidWorks
Corp\SolidWorks\SolidWorks.Interop.sldworks.dll'); % 2010 version
%h1     = NET.addAssembly('C:\Program Files\SolidWorks Corp\SolidWorks
(2)\SolidWorks.Interop.sldworks.dll') % Newer version
swApp = SolidWorks.Interop.sldworks.SldWorksClass;

% Make application visible
if ~(swApp.Visible)
    swApp.Visible = true;
end

Part   = swApp.OpenDoc6([pwd '\BasicRotor.SLDPRT'], 1, 0, [], 0,0);

% This allows geometries of less than 1mm
Part.SketchManager.AddToDB = true;
Part.SketchManager.DisplayWhenAdded = false;

%---------------------------------------------------------------------------

count.Curve = 1;

% Streamlines are generated
count.Streamline = 1;
for ii = 1:N_r
    if ii ~= 1 % General streamlines
        Part.InsertCurveFileBegin
        for jj = 1:length((Stream.z(ii,:)))
            boolstatus = Part.InsertCurveFilePoint(Stream.z(ii,jj),
Stream.r(ii,jj), zeros(size(Stream.z(ii,jj))));
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName         = ['Curve' num2str(count.Curve)];
        boolstatus        = Part.Extension.SelectByID2(CurveName,
'REFERENCECURVES', 0, 0, 0, false, 0, [], 0);
        CurveName         = ['Streamline' num2str(count.Streamline)];
        boolstatus        = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0,
0, true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.Streamline = count.Streamline + 1;

    else % Hub needs to be treated separately, zero points ignored
         % Previously used first row of Stream.r/z, currently uses a distinct
point set
        Part.InsertCurveFileBegin
        for jj = find([diff(HubProfile.r~=0) 0]==1):length((HubProfile.z))
            boolstatus      = Part.InsertCurveFilePoint(HubProfile.z(jj),
```

```
HubProfile.r(jj), zeros(size(HubProfile.z(jj))));
```

```
            temp.HubZ(1+jj-find([diff(HubProfile.r~=0) 0]==1))   =
HubProfile.z(jj);
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName         = ['Curve' num2str(count.Curve)];
        boolstatus        = Part.Extension.SelectByID2(CurveName,
'REFERENCECURVES', 0, 0, 0, false, 0, [], 0);
        CurveName         = ['Streamline' num2str(count.Streamline)];
        boolstatus        = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0,
0, true, false, CurveName);
        count.Curve       = count.Curve + 1;
        count.Streamline = count.Streamline + 1;

    end
end

Part.ClearSelection2(true)

% Calculations for the revolved boss outline
temp.Z = [min(min(Wedge.z(:,:,1))) max(max(Wedge.z(:,:,1)))];        % Extents
of wedge
temp.Z = [temp.HubZ(1) temp.Z(end)+0.1*diff(temp.Z)]; % 10% added end and
streamline beggining used
temp.Z = [temp.Z temp.Z(end) Stream.z(1,end)];   % Rest of sketch is added
temp.Y = [0 0 Stream.r(1,end) Stream.r(1,end)];
temp.X = zeros(size(temp.Z));

% Generate sketch to revolve
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0); % Plane for sketch
Part.SketchManager.InsertSketch(true); % Sketch is opened

for ii = 1:(length(temp.Z)-1)
    skSegment = Part.SketchManager.CreateLine(temp.Z(ii), temp.Y(ii),
temp.X(ii), temp.Z(ii+1), temp.Y(ii+1), temp.X(ii+1));
end

% Hub streamline is used in sketch
boolstatus = Part.Extension.SelectByID2('Streamline1', 'REFERENCECURVES', 0,
0, 0, true, 0, [], 0);
boolstatus = Part.SketchManager.SketchUseEdge2(false);


Part.SketchManager.InsertSketch(true) % Sketch is closed
Part.ClearSelection2(true)

%--------------------------------------------------------------------------
% Generate the rotated boss
boolstatus = Part.Extension.SelectByID2('Sketch1', 'SKETCH', 0, 0, 0, false,
0, [], 0);
boolstatus = Part.Extension.SelectByID2('Line1@Sketch1', 'EXTSKETCHSEGMENT',
0, 0, 0, true, 4, [], 0);
Part.FeatureManager.FeatureRevolve(6.28318530718, false, 0, 0, 0, true, true,
true);
```

```
Part.ClearSelection2(true)

% Part.ViewOrientationUndo()

%----------------------------------------------------------------------------
% PROFILES for computational volume  are created
count.Curve            = 1;
count.BladeGuideFront = 1;
count.BladeGuideRear   = 1;
for kk =1:size(Blade.z,3)
    for ii = 1:size(Blade.z,2)
        % Front side of blade
    Part.InsertCurveFileBegin
    for jj = 1:(length(Blade.z(:,ii,1))+1)
        if jj~=(length(Blade.z(:,ii,1))+1)
           boolstatus =
Part.InsertCurveFilePoint(Blade.z(jj,ii,kk),Blade.y(jj,ii,kk),Blade.x(jj,ii,k
k));
        else
           boolstatus =
Part.InsertCurveFilePoint(Blade.z(1,ii,kk),Blade.y(1,ii,kk),Blade.x(1,ii,kk))
;
        end
    end

    boolstatus = Part.InsertCurveFileEnd;

    % Name Change
    CurveName   = ['Curve' num2str(count.Curve)]; % old name in SW
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['ProfileCurve' num2str(count.BladeGuideFront)]; % new name
in SW
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve = count.Curve + 1;
    count.BladeGuideFront = count.BladeGuideFront +1;
    %ProfileCurves1-5 correspond to Blade 1
    end % for ii
end % for kk
disp('Blade guide curves generated')
Part.ClearSelection2(true);

% GUIDE CURVES for computational volume  are created
count.Curve            = 1;
count.BladeGuideFront = 1;
count.BladeGuideRear   = 1;
for kk =1:size(Blade.z,3)
    for ii = 1:2:size(Blade.x,1)
       % Front side of blade
    Part.InsertCurveFileBegin
    for jj = 1:length(Blade.y(ii,:,:))
           boolstatus =
Part.InsertCurveFilePoint(Blade.z(ii,jj,kk),Blade.y(ii,jj,kk),Blade.x(ii,jj,k
k));
```

```
end
```

```
            boolstatus = Part.InsertCurveFileEnd;

            % Name Change
    CurveName   = ['Curve' num2str(count.Curve)]; % old name in SW
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['GuideCurve' num2str(count.BladeGuideFront)]; % new name in
SW
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve       = count.Curve + 1;
    count.BladeGuideFront = count.BladeGuideFront +1;
            %GuideCurves11-30 correspond to Blade 1
    end % for ii
end % for kk
    disp('Blade guide curves generated')
Part.ClearSelection2(true);

% Generation of lofts using loops
count.BladeProfileCurve = 1;
count.BladeGuideFront = 1;
% Outer loop to iterate blades
for ii = 1:size(Blade.z,3)
    % Select the profile curves
    % First curve must be indicated 'false' in the function call
    boolstatus = Part.Extension.SelectByID2(['ProfileCurve'
num2str(count.BladeProfileCurve)], 'REFERENCECURVES', 0, 0, 0, false, 1, [],
0);
    count.BladeProfileCurve = count.BladeProfileCurve + 1;
    % The rest must be indicated 'true'
    for jj = 2:size(Blade.z,2)
        boolstatus = Part.Extension.SelectByID2(['ProfileCurve'
num2str(count.BladeProfileCurve)], 'REFERENCECURVES', 0, 0, 0, true, 1, [],
0);
        count.BladeProfileCurve = count.BladeProfileCurve + 1;
    end

    % Select the guide curves
    count.FnSelect = 2; % For this, stores the strange numbers used inside
the function call
    % NOTE: The '/2' below is to take into account the fact that only every
    % other guide curve was drawn from the data. This will need to be changed
    % if a different interval is used.
    for jj=1:(size(Blade.x, 1)/2)
        count.FnSelect = count.FnSelect + 4096;
        boolstatus = Part.Extension.SelectByID2(['GuideCurve'
num2str(count.BladeGuideFront)], 'REFERENCECURVES', 0, 0, 0, true,
count.FnSelect, [], 0);
        count.BladeGuideFront = count.BladeGuideFront + 1;
    end
    Part.FeatureManager.InsertProtrusionBlend (false, true, false, 1, 0, 0,
1, 1, true, true, false, 0, 0, 0, true, true, true);
end

%--------------------------------------------------------------------------
% Patterned Blades
```

```
% Rotational Axis
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true); % Sketch is opened
skSegment = Part.SketchManager.CreateLine(temp.Z(1), temp.Y(1), temp.X(1),
temp.Z(2), temp.Y(2), temp.X(2));
Part.SketchManager.InsertSketch(true); % Sketch is closed
Part.ClearSelection2(true);

for ii = 1:size(Blade.z,3)
    % First selection has a "false" rather than a true
    if ii==1
        boolstatus = Part.Extension.SelectByID2(['Loft' num2str(ii)],
'BODYFEATURE', 0, 0, 0, false, 4, [], 0);
    else
        boolstatus = Part.Extension.SelectByID2(['Loft' num2str(ii)],
'BODYFEATURE', 0, 0, 0, true, 4, [], 0);
    end
end
boolstatus = Part.Extension.SelectByID2('Sketch2', 'SKETCH', 0, 0, 0, true,
1, [], 0);
Part.FeatureManager.FeatureCircularPattern2(Blade.PassNo,    2*pi/Blade.PassNo,
false, 'NULL', true);

% Add fillets to the blades
% For maximum flexibility, must be done before slicing the ends off the
% hub because it adds fillets to the entire revolved solid. Any unwanted
% fillets will then be sliced off.
%%FilletRadius = 0.00254; % Fillet radius in meters
boolstatus = Part.Extension.SelectByID2('Revolve1', 'BODYFEATURE', 0, 0, 0,
false, 1, [], 0);
%myFeature = Part.FeatureManager.FeatureFillet(195, Blade.Fillet, 0, 0, [],
[], []);

% Check if the fillets were successfully created
if isempty(myFeature)
    fprintf('\n**WARNING**: Insert fillets failed.\nConsider reducing fillet
radius.\n\n')
end

%------------------------------------------------------------------------------
% Slice off the ends of the hub

% Calculations for where to make the first slice
% Goes from the extreme edge of the blade out past the flat end of the
revolved boss
% Left Edge is the extreme point of the blade plus 10% of the blade
% Right Edge calculation is based on the calculations for the edges of the
% revolving boss
% Top is an extra 10% from the revolving boss's height
MaxSpace = 0.1*(max(max(max(Blade.z))) - min(min(min(Blade.z)))); % How far
from the fillets the hub should extend on the 'max' side
temp.LeftEdge = max(max(max(Blade.z))) + Blade.Fillet + MaxSpace;
temp.RightEdge = max(max(Wedge.z(:,:,1))) + 0.15*(max(max(Wedge.z(:,:,1)))-
min(min(Wedge.z(:,:,1))));
temp.TopEdge = 1.1*Stream.r(1,end);
```

```
% Slice off one end of the revolved solid
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch( true);
Part.ClearSelection2( true);
Part.SketchManager.CreateCornerRectangle(temp.LeftEdge, temp.TopEdge, 0,
temp.RightEdge, 0, 0);
Part.ClearSelection2( true);
Part.SketchManager.InsertSketch( true);
Part.ClearSelection2( true);
boolstatus = Part.Extension.SelectByID2('Line1@Sketch2', 'EXTSKETCHSEGMENT',
0, 0, 0, true, 4, [], 0);
boolstatus = Part.Extension.SelectByID2('Sketch3', 'SKETCH', 0, 0, 0, true,
0, [], 0);
Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0, 0, true,
true);
Part.ClearSelection2( true);

% Calculations for the second slice
% Goes from the blade's other extreme edge past the pointed end of the
% revolved boss
% Right Edge is the extreme point of the blade plus fillet radius plus 10% of
the blade
% Left Edge goes to the farthest edge of the stream curves, which should be
beyond
% the rotated boss
% Top remains the same from the other cut
MinSpace = 0.1*(max(max(max(Blade.z))) - min(min(min(Blade.z)))); % How far
from the fillets the hub should extend on the 'min' side
temp.RightEdge = min(min(min(Blade.z))) - Blade.Fillet - MinSpace;
temp.LeftEdge = min(min(Wedge.z(:,:,1)));

% Slice off the other end of the revolved solid
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch( true);
Part.ClearSelection2( true);
Part.SketchManager.CreateCornerRectangle(temp.LeftEdge, temp.TopEdge, 0,
temp.RightEdge, 0, 0);
Part.ClearSelection2( true);
Part.SketchManager.InsertSketch( true);
Part.ClearSelection2( true);
boolstatus = Part.Extension.SelectByID2('Line1@Sketch2', 'EXTSKETCHSEGMENT',
0, 0, 0, true, 4, [], 0);
boolstatus = Part.Extension.SelectByID2('Sketch4', 'SKETCH', 0, 0, 0, true,
0, [], 0);
Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0, 0, true,
true);
Part.ClearSelection2( true);

%--------------------------------------------------------------------------
% Sketched Rectangle to be used in Revolved cut of BLADES

% Centerline or rotation axis is placed
```

```
temp.Z = [min(min(Wedge.z(:,:,1))) max(max(Wedge.z(:,:,1)))];        % Extents
of wedge
temp.Z = [temp.Z(1)-0.1*diff(temp.Z) temp.Z(end)+0.1*diff(temp.Z)]; % 10%
added to begining and end
temp.Y = [0 0];
temp.X = temp.Y;

% Sketch of rotational axis is placed
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true); % Sketch is opened
skSegment = Part.SketchManager.CreateLine(temp.Z(1), temp.Y(1), temp.X(1),
temp.Z(2), temp.Y(2), temp.X(2));
Part.SketchManager.InsertSketch(true); % Sketch is closed
Part.ClearSelection2(true);

% Top box at a specific height (r = 0.14351 m) to create desired blade tip
gap (00/1000 inch) is placed
temp.Z = [temp.Z(1) temp.Z(1) temp.Z(end) temp.Z(end) temp.Z(1)];
%temp.Z = [Stream.z(end,1) temp.Z(1) temp.Z(1) temp.Z(end) temp.Z(end)
Stream.z(end,end)]
R = max(max(max(Blade.r)));
temp.Y = [0.14351 2*R 2*R 0.14351 0.14351];
%temp.Y = [0.9*Stream.r(end,1) 1.1*R 1.1*R 0.9*Stream.r(end,end)
0.9*Stream.r(end,1)]
%temp.Y = [0.9*Stream.r(end,1) 0.9*Stream.r(end,1) 1.1*R 1.1*R
0.9*Stream.r(end,end) 0.9*Stream.r(end,end)]
temp.X = zeros(size(temp.Y));
clear R

boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true);
for ii = 1:(length(temp.Z)-1)
    skSegment = Part.SketchManager.CreateLine(temp.Z(ii), temp.Y(ii),
temp.X(ii), temp.Z(ii+1), temp.Y(ii+1), temp.X(ii+1));
end
boolstatus = Part.SketchManager.SketchUseEdge2(false);
Part.SketchManager.InsertSketch(true); % Sketch is closed
Part.ClearSelection2(true);

%--------------------------------------------------------------------------
% Sketched Rectangle to be used in revolved cut of AIRWEDGE

% Centerline or rotation axis is placed
temp1.Z = [min(min(Wedge.z(:,:,1))) max(max(Wedge.z(:,:,1)))];        %
Extents of wedge
temp1.Z = [temp1.Z(1)-0.1*diff(temp1.Z) temp1.Z(end)+0.1*diff(temp1.Z)]; %
10% added to begining and end
temp1.Y = [0 0];
temp1.X = temp1.Y;

% Sketch of rotational axis is placed
boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true); % Sketch is opened
```

```
skSegment = Part.SketchManager.CreateLine(temp1.Z(1), temp1.Y(1), temp1.X(1),
temp1.Z(2), temp1.Y(2), temp1.X(2));
Part.SketchManager.InsertSketch(true); % Sketch is closed
Part.ClearSelection2(true);

% Top box is placed
temp1.Z = [temp1.Z(1) temp1.Z(1) temp1.Z(end) temp1.Z(end)];
%R = max(max(max(Blade.r)));
R = 0.1440688;
%temp1.Y = [ Stream.r(end,1) 2*R 2*R Stream.r(end,end)];
temp1.Y = [ R 2*R 2*R R];
temp1.X = zeros(size(temp1.Y));
clear R

boolstatus = Part.Extension.SelectByID2('Front Plane', 'PLANE', 0, 0, 0,
false, 0, [], 0);
Part.SketchManager.InsertSketch(true);
for ii = 1:(length(temp1.Z)-1)
    skSegment = Part.SketchManager.CreateLine(temp1.Z(ii), temp1.Y(ii),
temp1.X(ii), temp1.Z(ii+1), temp1.Y(ii+1), temp1.X(ii+1));
end
%boolstatus = Part.Extension.SelectByID2('Streamline12', 'REFERENCECURVES',
0, 0, 0, true, 0, [], 0); % Bounding streamline is used
boolstatus = Part.SketchManager.SketchUseEdge2(false);
Part.SketchManager.InsertSketch(true); % Sketch is closed
Part.ClearSelection2(true);

%--------------------------------------------------------------------------
% Blades Trimmed
boolstatus = Part.Extension.SelectByID2('Line1@Sketch5', 'EXTSKETCHSEGMENT',
-0.1874958079273, 0, 0, true, 4, [], 0);
boolstatus = Part.Extension.SelectByID2('Sketch6', 'SKETCH', 0, 0, 0, true,
0, [], 0);
Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0, 0, true,
true);
Part.ClearSelection2(true);

% Save model at this point as a parasolid
longstatus = Part.SaveAs3([pwd '\BladeHub_Full.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\BladeHub_Full.SLDPRT'], 0, 2);
disp('Full rotor generated. BladeHubFull.X_T saved.')

%--------------------------------------------------------------------------
% Reduce rotor to only the bit that sits inside the air wedge

% Generate the air wedge
% Guide Curves for computational volume  are created
count.Curve            = 1;
count.WedgeGuideFront = 1;
count.WedgeGuideRear   = 1;
for ii = 1:size(Wedge.z,2)
    % Front side of wedge
    Part.InsertCurveFileBegin;
    for jj = 1:length(Wedge.z(:,ii,1))
```

```
        boolstatus = 
Part.InsertCurveFilePoint(Wedge.z(jj,ii,1),Wedge.y(jj,ii,1),Wedge.x(jj,ii,1))
;
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0, 
0, 0, false, 0, [], 0);
    CurveName   = ['FrontWedgeGuideCurve' num2str(count.WedgeGuideFront)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true, 
false, CurveName);
    count.Curve        = count.Curve + 1;
    count.WedgeGuideFront = count.WedgeGuideFront +1;

    % Back side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length(Wedge.z(:,ii,1))
        boolstatus = 
Part.InsertCurveFilePoint(Wedge.z(jj,ii,2),Wedge.y(jj,ii,2),Wedge.x(jj,ii,2))
;
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0, 
0, 0, false, 0, [], 0);
    CurveName   = ['RearWedgeGuideCurve' num2str(count.WedgeGuideRear)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true, 
false, CurveName);
    count.Curve       = count.Curve + 1;
    count.WedgeGuideRear = count.WedgeGuideRear +1;
end % for ii
disp('Wedge guide curves generated')

% Streamwise Curves for computational volume are generated
count.WedgeStreamFront = 1;
count.WedgeStreamRear   = 1;
for ii = 1:size(Wedge.z,1)
    % Front side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(ii,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(ii,jj,1)),
(Wedge.y(ii,jj,1)), (Wedge.x(ii,jj,1)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0, 
0, 0, false, 0, [], 0);
    CurveName   = ['FrontWedgeStreamCurve' num2str(count.WedgeStreamFront)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true, 
false, CurveName);
```

```
    count.Curve       = count.Curve + 1;
    count.WedgeStreamFront = count.WedgeStreamFront +1;

    % Back side of wedge
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(ii,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(ii,jj,2)),
(Wedge.y(ii,jj,2)), (Wedge.x(ii,jj,2)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['RearWedgeStreamCurve' num2str(count.WedgeStreamRear)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.WedgeStreamRear = count.WedgeStreamRear +1;
end
disp('Wedge stream curves generated')

% Front Surface of gas computational volume is created
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.WedgeStreamFront-1)
    CurveName   = ['FrontWedgeStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.WedgeGuideFront-1)
    CurveName   = ['FrontWedgeGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0);

% Rear Surface of gas computational volume is created
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.WedgeStreamRear-1)
    CurveName   = ['RearWedgeStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.WedgeGuideRear-1)
```

```
    CurveName   = ['RearWedgeGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0);

% Top surface for the computational volume is created
count.CaseStream = 1;
for ii = 1:2;  % Stream direction curves are placed
    Part.InsertCurveFileBegin
    for jj = 1:length((Wedge.z(end,:,1)))
        boolstatus = Part.InsertCurveFilePoint((Wedge.z(end,jj,ii)),
(Wedge.y(end,jj,ii)), (Wedge.x(end,jj,ii)));
    end
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['CaseStreamCurve' num2str(count.CaseStream)]; boolstatus
    = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.CaseStream = count.CaseStream + 1;
end

count.CaseGuide = 1;
for ii = 1:length(Wedge.z(end,:,1)) % Guide direction curves are placed
    Part.InsertCurveFileBegin
    % First point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,1),
Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    % Intermediate point
    [TH(1),R(1)] = cart2pol(Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    [TH(2),R(2)] = cart2pol(Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    [temp.Y temp.X temp.Z] = pol2cart(mean(TH), mean(R),
mean([Wedge.z(end,ii,1) Wedge.z(end,ii,2)]));
    boolstatus = Part.InsertCurveFilePoint(temp.Z, temp.Y, temp.X);
    % Last point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,2),
Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['CaseGuideCurve' num2str(count.CaseGuide)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve     = count.Curve + 1;
    count.CaseGuide = count.CaseGuide + 1;

end
```

```
% Casing surface of computational volume is created
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
for ii = 1:(count.CaseStream-1)
    CurveName   = ['CaseStreamCurve' num2str(ii)];
    if ii == 1; BoolVal = false;  else BoolVal = true; end;
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, BoolVal, 1, [], 0);
end % for ii

% Guide curves are chosen
count.Mark = 4098;
for ii = 1:(count.CaseGuide-1)
    CurveName   = ['CaseGuideCurve' num2str(ii)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, true, count.Mark, [], 0);
    count.Mark = count.Mark + 4096;
end % for ii
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0);

% Front bound surface of computational volume is created
% This is the same as for the sides of the control volume but just the front
and back
count.BoundEnd         = 1;
for ii = [1 size(Wedge.z,2)]
    for kk = 1:2 % Sides of bound surface are created
        Part.InsertCurveFileBegin
        for jj = 1:length(Wedge.z(:,ii,1))
            boolstatus =
Part.InsertCurveFilePoint(Wedge.z(jj,ii,kk),Wedge.y(jj,ii,kk),Wedge.x(jj,ii,k
k));
        end
        boolstatus = Part.InsertCurveFileEnd;

        % Change name to something meaningful
        CurveName   = ['Curve' num2str(count.Curve)];
        boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES',
0, 0, 0, false, 0, [], 0);
        CurveName   = ['BoundEndCurve' num2str(count.BoundEnd)];
        boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0,
true, false, CurveName);
        count.Curve     = count.Curve + 1;
        count.BoundEnd = count.BoundEnd +1;
    end % for kk

    % The top curves of the bound surfaces are created
    Part.InsertCurveFileBegin;
    % First point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,1),
Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    % Intermediate point
    [TH(1),R(1)] = cart2pol(Wedge.y(end,ii,1), Wedge.x(end,ii,1));
    [TH(2),R(2)] = cart2pol(Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    [temp.Y temp.X temp.Z] = pol2cart(mean(TH), mean(R),
mean([Wedge.z(end,ii,1) Wedge.z(end,ii,2)]));
```

```
    boolstatus = Part.InsertCurveFilePoint(temp.Z, temp.Y, temp.X);
    % Last point
    boolstatus = Part.InsertCurveFilePoint(Wedge.z(end,ii,2),
Wedge.y(end,ii,2), Wedge.x(end,ii,2));
    boolstatus = Part.InsertCurveFileEnd;

    % Change name to something meaningful
    CurveName   = ['Curve' num2str(count.Curve)];
    boolstatus = Part.Extension.SelectByID2(CurveName, 'REFERENCECURVES', 0,
0, 0, false, 0, [], 0);
    CurveName   = ['BoundEndCurve' num2str(count.BoundEnd)];
    boolstatus = Part.SelectedFeatureProperties(0, 0, 0, 0, 0, 0, 0, true,
false, CurveName);
    count.Curve      = count.Curve + 1;
    count.BoundEnd = count.BoundEnd +1;

end % for ii

% Lofted ends are used as filled surface macro not working
Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
boolstatus = Part.Extension.SelectByID2('BoundEndCurve1', 'REFERENCECURVES',
0, 0, 0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve2', 'REFERENCECURVES',
0, 0, 0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve3', 'REFERENCECURVES',
0, 0, 0, true, 4098, [], 0);
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0);

Part.ClearSelection2(true) % Selections are cleared
% Streamwise curves are chosen
boolstatus = Part.Extension.SelectByID2('BoundEndCurve4', 'REFERENCECURVES',
0, 0, 0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve5', 'REFERENCECURVES',
0, 0, 0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('BoundEndCurve6', 'REFERENCECURVES',
0, 0, 0, true, 4098, [], 0);
Part.InsertLoftRefSurface2( false, true, false, 1, 0, 0);
Part.ClearSelection2(true); % Selections are cleared

% Solid is sewn together
boolstatus = Part.Extension.SelectByID2('Surface-Loft1', 'SURFACEBODY', 0, 0,
0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft2', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft3', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft4', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Loft5', 'SURFACEBODY', 0, 0,
0, true,  1, [], 0);
myFeature = Part.FeatureManager.InsertSewRefSurface(true, true, true,
0.00011684, 0.0001);
Part.ClearSelection2(true); % Selections are cleared

% Keep only the solid areas enclosed by both the hub and the wedge
```

```
boolstatus = Part.Extension.SelectByID2('Cut-Revolve3', 'SOLIDBODY', 0, 0, 0,
false, 2, [], 0);
boolstatus = Part.Extension.SelectByID2('Surface-Knit1', 'SOLIDBODY', 0, 0,
0, true, 2, [], 0);
myFeature = Part.FeatureManager.InsertCombineFeature(15901, [], []);

% Save the wedge as a parasolid as well
longstatus = Part.SaveAs3([pwd '\BladeHub_Wedge_CutOut.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\BladeHub_Wedge_CutOut.SLDPRT'], 0, 2);
disp('Rotor wedge generated. BladeHub_Wedge_Cutout.X_T saved.')

%    ---------------------------------------------------------------------
% Save the air wedge with fillets
% Delete the combine feature,the cut that removed the ends of the
% blades, and the cut that removed the ends of the hub
boolstatus = Part.Extension.SelectByID2('Combine1', 'BODYFEATURE', 0, 0, 0,
false, 0, [], 0);
%boolstatus = Part.Extension.SelectByID2('Cut-Revolve3', 'BODYFEATURE', 0, 0,
0, true, 0, [], 0);
boolstatus = Part.Extension.SelectByID2('Cut-Revolve1', 'BODYFEATURE', 0, 0,
0, true, 0, [], 0);
boolstatus = Part.Extension.SelectByID2('Cut-Revolve2', 'BODYFEATURE', 0, 0,
0, true, 0, [], 0);
Part.EditDelete;

% Revolved cut is performed (this is for SW 2010 version)
boolstatus = Part.Extension.SelectByID2('Line1@Sketch2', 'EXTSKETCHSEGMENT',
0, 0, 0, false, 4, [], 0); % Axis of revolution
boolstatus = Part.Extension.SelectByID2('Sketch8', 'SKETCH', 0, 0, 0, true,
2, [], 0);                    % Sketch for cut
myFeature = Part.FeatureManager.FeatureRevolveCut(6.28318530718, false, 0, 0,
0, true, true);           % Revolved cut

Part.ClearSelection2(true);

% Subtract the rotor from the air wedge
boolstatus = Part.Extension.SelectByID2('Cut-Revolve4', 'SOLIDBODY', 0, 0, 0,
false, 1, [], 0);
%boolstatus = Part.Extension.SelectByID2('Cut-Revolve4[2]', 'SOLIDBODY', 0,
0, 0, false, 1, [], 0);
boolstatus = Part.Extension.SelectByID2('Cut-Revolve3', 'SOLIDBODY', 0, 0, 0,
true, 2, [], 0);
%boolstatus = Part.Extension.SelectByID2('Cut-Revolve4[1]', 'SOLIDBODY', 0,
0, 0, true, 2, [], 0);
Part.FeatureManager.InsertCombineFeature(15902, [], []);

% Save model as a parasolid
longstatus = Part.SaveAs3([pwd '\AirWedge_Fillets.X_T'], 0, 0);
longstatus = Part.SaveAs3([pwd '\AirWedge_Fillets.SLDPRT'], 0, 2);
disp('Air wedge with fillets generated.  AirWedge_Fillets.X_T saved.')

% This returns the program to normal operation
Part.SketchManager.AddToDB = false;
Part.SketchManager.DisplayWhenAdded = true;
```

XXIII. FLUIDANALYSIS.M

```
% Runs the ANSYS script to refresh the air fluid analysis
% Input parameters:
%   -OutletPressure: Specifies the outlet pressure in atmospheres
%   -AngularVelocity: Specifies the angular velocity in rpm
%       NOTE: For a typical rotor analysis, must be entered as a
%       negative value (i.e. -30000)
%   -PassNo: Number of blade passages, used to calculate to total mass
%       flow.  Should be identical to Blade.PassNo.
% Returns structure: "outputs", containing:
%   -effTT: Total-to-total isentropic efficiency
%   -pTotalOut: Total Pressure, Outlet
%   -pTotalOutUnits: Units of pTotalOut
%   -pTotalIn: Total Pressure, Inlet
%   -pTotalInUnits: Units of pTotalIn
%   -mFlowIn: Total mass flow at inlet for the OVERALL rotor
%   -mFlowInUnits: Units of mFlowIn
%   -mFlowOut: Total mass flow at outlet for the OVERALL rotor
%   -mFlowOutUnits: Units of mFlowIn

function outputs = FluidAnalysis(OutletPressure, AngularVelocity, PassNo,
RunNo)

% Write Blade.OutletPressure to a file for ANSYS to read in.  Overwrite file
if it exists.
filePath = 'E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\';
try
    file = fopen([filePath 'InputParams.dat'], 'wt');
    fprintf(file, '%f\n%f', OutletPressure, AngularVelocity);
% Only catch exceptions so that the file can properly be closed if there is
an error
catch err
    fclose(file);
    rethrow(err)
end
fclose(file);
disp(['Outlet Pressure saved to ' filePath 'InputParams.dat'])

fprintf('\nBeginning analysis: %g [atm] outlet pressure, %g [rpm] angular
velocity.\n', OutletPressure, AngularVelocity)
timestamp = clock;
% Display the current time
fprintf('Start time %d:%02d\n', timestamp(4), timestamp(5))
fprintf('Refreshing and rerunning analysis....\n')
timeStart = tic;

%if RunNo == 1
if RunNo == 0
    ! "C:\Program Files\ANSYS Inc\v140\Framework\bin\Win64\RunWB2" -F
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\WorkingProject
.wbpj" -R
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\UpdateProject1
.py" -X
else
```

```
    ! "C:\Program Files\ANSYS Inc\v140\Framework\bin\Win64\RunWB2" -F
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\WorkingProject
.wbpj" -R
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\UpdateProject2
.py" -X
end
disp('Analysis complete.')
% Display Elapsed Time
% If time is longer than one hour, display in hours, minutes, and seconds.
% Otherwise, just minutes and seconds.
timeEnd = toc(timeStart);
fprintf('Elapsed time is ');
if (timeEnd/3600) > 2
    fprintf('%d hours ', floor(timeEnd/3600))
elseif (timeEnd/3600) > 1
    fprintf('%d hour ', floor(timeEnd/3600))
end
fprintf('%d minutes and %f seconds.\n', floor(rem(timeEnd,3600)/60),
rem(timeEnd,60))

% Read 'SavedOutput.dat'
fprintf('\n\nReading analyis output...\n')
outputs = ReadAnsysData([filePath 'SavedOutput.dat']);

% Calculate the overall mass flow and power from the single-passage mass flow
and power
outputs.mFlowIn = outputs.mFlowIn*PassNo;
outputs.mFlowOut = outputs.mFlowOut*PassNo;
outputs.power = outputs.power*PassNo;

fprintf('Total-to-total isentropic efficiency: %g\n', outputs.effTT);
fprintf('Total Pressure, Outlet: %g %s\n', outputs.pTotalOut,
outputs.pTotalOutUnits);
fprintf('Total Pressure, Inlet: %g %s\n', outputs.pTotalIn,
outputs.pTotalInUnits);
fprintf('Total Pressure Ratio: %g\n', outputs.pTotalOut/outputs.pTotalIn);
fprintf('Total Mass Flow, Inlet: %g %s\n', outputs.mFlowIn,
outputs.mFlowInUnits);
fprintf('Total Mass Flow, Outlet: %g %s\n', outputs.mFlowOut,
outputs.mFlowOutUnits);
fprintf('Power (all blades): %g %s\n', outputs.power, outputs.powerUnits);
```

```
import os
```

XXIV. UPDATEPROJECT1.PY

```
# Single variable to store the file path for the saved output
filePath = "E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\\"

# Erase SavedOutput.dat so that MATLAB can tell if the run failed
open(filePath + "SavedOutput.dat", "w").close()

# The following code is from ANSYS Tech Support
# Define all Inputs

DM_macro_file =
r"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\DM_CAD_Refres
h.js"
# Has all the required operation to Refresh the CAD.

# Read the DM_macro_file to get all the commands
DMscript = open(DM_macro_file, "r")
DMscriptcommand=DMscript.read()
DMscript.close()

# Open DM Session in batch to Refresh the CAD file
system1 = GetSystem(Name="CFX")
geometry1 = system1.GetContainer(ComponentName="Geometry")
geometry1.Edit(Interactive=False) # batch mode

# Send the DM_macro_file to change the CAD File
geometry1.SendCommand(Command = DMscriptcommand)

# Exit DM session
geometry1.Exit()

# Update the cell state
geometry1.Update()
# End ANSYS Tech Support Code

#-----------------------------------------------------------------------------
---------------------------------------------------------------

# Update the new Output Pressure parameter

# Open the file holding the input parameters
try:
    file = open(filePath + "InputParams.dat")
    # Read the outlet pressure in atmospheres
    outPressure = file.readline()
    angularVelocity = file.readline()
finally:
    file.close()

# Remove whitespace from the strings (readline() leaves "\n" at the end of
lines)
```

```
outPressure.strip()
```

```
angularVelocity.strip()

designPoint1 = Parameters.GetDesignPoint(Name="0")
parameter1 = Parameters.GetParameter(Name="P1")
designPoint1.SetParameterExpression(
    Parameter=parameter1,
    Expression=(outPressure + " [atm]"))

parameter5 = Parameters.GetParameter(Name="P5")
designPoint1.SetParameterExpression(
    Parameter=parameter5,
    Expression=(angularVelocity + " [rev min^-1]"))

# Update the rest of the project
#Update()
component1 = system1.GetComponent(Name="Mesh")
component1.Update(AllDependencies=True)
component2 = system1.GetComponent(Name="Setup")
component2.Update(AllDependencies=True)
solution1 = system1.GetContainer(ComponentName="Solution")
solution1.Edit()         # This is the critical line that shows the solution
graph.
                         # If not for this line, this whole section could be
replaced with "Update()".
                         # Note that this will only correctly display the plot
if there is already preexisting saved run data.
component3 = system1.GetComponent(Name="Solution")
component3.Update(Force=True)
solution1.Exit()
component4 = system1.GetComponent(Name="Results")
component4.Update(AllDependencies=True)

# Save the project
Save()

#-----------------------------------------------------------------------------
----------------------------------------------------------------

# Calculate and save all output parameters

# Run the Gas Compressor Performance macro calculator
results1 = system1.GetContainer(ComponentName="Results")
results1.SendCommand(Command="!compressorPerform( \"Inlet\", \"Outlet\",
\"Default Domain Default\", \"x\", \"" + angularVelocity + " [rev min^-1]\",
\"1\", \"1 [atm]\", \"0.05 [m]\", \"1.4\" );")

# Extract the output parameters
effTT = Parameters.GetParameter(Name="P2").Value        # Total-to-total
isentropic efficiency
pTotalOut = Parameters.GetParameter(Name="P3").Value    # Total Output
Pressure
pTotalIn = Parameters.GetParameter(Name="P4").Value     # Total Input
Pressure
mFlowIn = Parameters.GetParameter(Name="P6").Value      # Mass Flow at Inlet
mFlowOut = Parameters.GetParameter(Name="P7").Value     # Mass Flow at Outlet
```

```
Power = Parameters.GetParameter(Name="P8").Value         # Power

# Save the output parameters to a file

# Use a try-finally block to ensure that the file is closed if an exception
is thrown while trying to write
# Exceptions are not caught
try:
    file = open(filePath + "SavedOutput.dat", "w")
    file.write("Total-to-total isentropic efficiency: ")
    try:
        file.write(effTT.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Total Pressure, Outlet: ")
    try:
        file.write(pTotalOut.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Total Pressure, Inlet: ")
    try:
        file.write(pTotalIn.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Mass Flow, Inlet: ")
    try:
        file.write(mFlowIn.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Mass Flow, Outlet: ")
    try:
       file.write(mFlowOut.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Power (all blades): ")
    try:
        file.write(Power.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
finally:
    file.close()
```

XXV. DM_CAD_REFRESH.JS

```
var NodeName = "Import1"; // Name of the import feature

// Loop through the existing features and select the required one
var count = ag.fm.FeatureCount;
for (var i =0; i < count; i++) {

    var current = ag.fm.feature(i);
    var Name = current.Name;

    if (Name.toLowerCase() == NodeName.toLowerCase()) { // found the match
    current.Refresh = 1; // Refresh the Geometry
    break;
    }
}
agb.regen();
```

XXVI. UPDATEPROJECT2.PY

```
import os

# Single variable to store the file path for the saved output
filePath = "E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\\"

# Erase SavedOutput.dat so that MATLAB can tell if the run failed
open(filePath + "SavedOutput.dat", "w").close()

system1 = GetSystem(Name="CFX")

# Update the new Output Pressure parameter

# Open the file holding the input parameters
try:
    file = open(filePath + "InputParams.dat")
    # Read the outlet pressure in atmospheres
    outPressure = file.readline()
    angularVelocity = file.readline()
finally:
    file.close()

# Remove whitespace from the strings (readline() leaves "\n" at the end of
lines)
outPressure.strip()
angularVelocity.strip()

designPoint1 = Parameters.GetDesignPoint(Name="0")
parameter1 = Parameters.GetParameter(Name="P1")
designPoint1.SetParameterExpression(
    Parameter=parameter1,
    Expression=(outPressure + " [atm]"))

parameter5 = Parameters.GetParameter(Name="P5")
designPoint1.SetParameterExpression(
    Parameter=parameter5,
    Expression=(angularVelocity + " [rev min^-1]"))

# Update the rest of the project
#Update()
component2 = system1.GetComponent(Name="Setup")
component2.Update(AllDependencies=True)
solution1 = system1.GetContainer(ComponentName="Solution")
solution1.Edit()         # This is the critical line that shows the solution
graph.
                         # If not for this line, this whole section could be
replaced with "Update()".  Note that this will only correctly display the
plot if there is already preexisting saved run data.
component3 = system1.GetComponent(Name="Solution")
component3.Update(Force=True)
solution1.Exit()
component4 = system1.GetComponent(Name="Results")
```

```
component4.Update(AllDependencies=True)
```

```
# Save the project
Save()

#---------------------------------------------------------------------------
-----------------------------------------------------------------

# Calculate and save all output parameters

# Run the Gas Compressor Performance macro calculator
results1 = system1.GetContainer(ComponentName="Results")
results1.SendCommand(Command="!compressorPerform( \"Inlet\", \"Outlet\",
\"Default Domain Default\", \"x\", \"" + angularVelocity + " [rev min^-1]\",
\"1\", \"1 [atm]\", \"0.05 [m]\", \"1.4\" );")

# Extract the output parameters
effTT = Parameters.GetParameter(Name="P2").Value        # Total-to-total
isentropic efficiency
pTotalOut = Parameters.GetParameter(Name="P3").Value    # Total Output
Pressure
pTotalIn = Parameters.GetParameter(Name="P4").Value     # Total Input
Pressure
mFlowIn = Parameters.GetParameter(Name="P6").Value      # Mass Flow at Inlet
mFlowOut = Parameters.GetParameter(Name="P7").Value     # Mass Flow at Outlet
Power = Parameters.GetParameter(Name="P8").Value        # Power

# Save the output parameters to a file

# Use a try-finally block to ensure that the file is closed if an exception
is thrown while trying to write
# Exceptions are not caught
try:
    file = open(filePath + "SavedOutput.dat", "w")
    file.write("Total-to-total isentropic efficiency: ")
    try:
        file.write(effTT.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Total Pressure, Outlet: ")
    try:
        file.write(pTotalOut.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Total Pressure, Inlet: ")
    try:
        file.write(pTotalIn.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Mass Flow, Inlet: ")
    try:
        file.write(mFlowIn.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Mass Flow, Outlet: ")
    try:
```

```
       file.write(mFlowOut.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
    file.write("Single-Passage Power (all blades): ")
    try:
        file.write(Power.ToString() + "\n")
    except TypeError:
        file.write("ERROR\n")
finally:
    file.close()
```

XXVII. READANSYSDATA.M

```
% Reads data exported from automated ANSYS CFX analysis
% Returns the structure "outputs", which contains:
%   -effTT: Total-to-total isentropic efficiency
%   -pTotalOut: Total Pressure, Outlet
%   -pTotalOutUnits: Units of pTotalOut
%   -pTotalIn: Total Pressure, Inlet
%   -pTotalInUnits: Units of pTotalIn
%   -mFlowIn: Mass flow at inlet for a SINGLE blade passage
%   mFlowInUnits: Units of mFlowIn
%   -mFlowOut: Mass flow at outlet for a SINGLE blade passage
%   -mFlowOutUnits: Units of mFlowIn

function outputs = ReadAnsysData(fileName)

pause(60);

raw = fileread(fileName);        % Read the entire file into a string

%---------------------------------------------------------------------------

% Throw an exception if file is blank.  This indicates the run did not
complete.
if strcmp(raw, '')
    err = MException('MATLAB:NoANSYSOutput', 'ERROR: No available ANSYS
output.\nANSYS analysis likely did not complete.');
    throw(err)
end

% Parse the data to check formatting and to read values
% If at any point the expected input is not found, throw an exception

% Read "Total-to-total isentropic efficiency: ####"
[temp, remainder] = strtok(raw);
contentsCheck(temp, 'Total-to-total');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'isentropic');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'efficiency:');

[temp, remainder] = strtok(remainder);
% Write the value as "Not a Number" if the value simply reads "ERROR"
if strcmp(temp, 'ERROR')
    outputs.effTT = NaN;
    disp('WARNING: Error computing Total-to-Total Isentropic Efficiency.')
else
% Otherwise, just read the values as expected
    outputs.effTT = str2double(temp);
    if isnan(outputs.effTT)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
```

```
contents.\nTotal-to-total isentropic efficiency not found in file.');
```

```
        throw(err)
    end
end


% Read "Total Pressure, Outlet: ####"
[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Total');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Pressure,');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Outlet:');

[temp, remainder] = strtok(remainder);
if strcmp(temp, 'ERROR')
    outputs.pTotalOut = NaN;
    outputs.pTotalOutUnits = '';
    disp('WARNING: Error computing Total Pressure, Outlet')
else
    outputs.pTotalOut = str2double(temp);
    if isnan(outputs.pTotalOut)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nTotal outlet pressure not found in file.');
        throw(err)
    end
    [outputs.pTotalOutUnits, remainder] = readUnits(remainder);
end


% Read "Total Pressure, Inlet: ####"
[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Total');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Pressure,');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Inlet:');

[temp, remainder] = strtok(remainder);
if strcmp(temp, 'ERROR')
    outputs.pTotalIn = NaN;
    outputs.pTotalInUnits = '';
    disp('WARNING: Error computing Total Pressure, Inlet')
else
    outputs.pTotalIn = str2double(temp);
    if isnan(outputs.pTotalIn)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nTotal inlet pressure not found in file.');
        throw(err)
    end
    [outputs.pTotalInUnits, remainder] = readUnits(remainder);
end
```

```
% Read "Single-Passage Mass Flow, Inlet: ####"
```

```
[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Single-Passage');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Mass');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Flow,');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Inlet:');

[temp, remainder] = strtok(remainder);
if strcmp(temp, 'ERROR')
    outputs.mFlowIn = NaN;
    outputs.mFlowInUnits = '';
    disp('WARNING: Error computing Single-Passage Mass Flow, Inlet')
else
    outputs.mFlowIn = str2double(temp);
    if isnan(outputs.mFlowIn)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nInlet mass flow not found in file.');
        throw(err)
    end
    [outputs.mFlowInUnits, remainder] = readUnits(remainder);
end


% Read "Single-Passage Mass Flow, Outlet: ####"
[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Single-Passage');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Mass');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Flow,');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Outlet:');

[temp, remainder] = strtok(remainder);
if strcmp(temp, 'ERROR')
    outputs.mFlowOut = NaN;
    outputs.mFlowOutUnits = '';
    disp('WARNING: Error computing Single-Passage Mass Flow, Outlet')
else
    outputs.mFlowOut = str2double(temp);
    if isnan(outputs.mFlowOut)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nOutlet mass flow not found in file.');
        throw(err)
    end
    [outputs.mFlowOutUnits, remainder] = readUnits(remainder);
end
```

```
% Read "Single-Passage Power (all blades): ####"
[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Single-Passage');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'Power');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, '(all');

[temp, remainder] = strtok(remainder);
contentsCheck(temp, 'blades):');

[temp, remainder] = strtok(remainder);
if strcmp(temp, 'ERROR')
    outputs.power = NaN;
    outputs.powerUnits = '';
    disp('WARNING: Error computing Power (all blades)')
else
    outputs.power = str2double(temp);
    if isnan(outputs.power)
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nPower (all blades) not found in file.');
        throw(err)
    end
    outputs.powerUnits = readUnits(remainder);
end


%--------------------------------------------------------------------------

% Private function to check file contents and throw an exception if there's a
mismatch
function contentsCheck(read, expected)
if ~strcmp(read, expected)
    err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nExpected %s, found %s.', expected, read);
    throw(err)
end

%--------------------------------------------------------------------------

% Private function that reads in units from the string, even if the units
% contain spaces.  Examples include [Pa], [kg s^-1]
function [unit remainder] = readUnits(input)

[unit remainder] = strtok(input);
% Make sure that this might actually be a unit by looking for an open bracket
at the beginning
if unit(1) ~= '['
    err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nExpected a measurement unit, found %s.', unit);
    throw(err)
end
```

```
% Iterate until you get to a closed bracket. This signifies the end of the
unit.
while unit(size(unit)) ~= ']'
    [temp remainder] = strtok(remainder);
    % If temp is empty, the end of the file was reached without finding a
closed bracket.  Throw an exception.
    if strcmp(temp, '')
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nExpected a measurement unit, found %s.', unit);
        throw(err);
    end
    % If there is a bracket somewhere in 'unit' but not at the end, also
throw an exception.
    if ~isempty(strfind(unit, ']'))
        err = MException('MATLAB:FormatMismatch', 'Unexpected file
contents.\nExpected a measurement unit, found %s.', unit);
        throw(err);
    end
    unit = [unit ' ' temp];
end
```

XXVIII. ARCHIVERUNDATA.M

```
% Copy all files relating to the latest run
% Essentially copies all the .m files, saves the run data into an Excel
% spreadsheet, and archives/wipes clean the ANSYS working project

% NOTE: This script assumes that nothing happens to change the files in the
% current working directory or file desinations while this runs.

fprintf('\nFile archiving/storage in progress. Do not alter files in working
directory.\n')

% Generate the base folder name which numbers will be appended to
baseFolderName = [pwd '\CompletedAnalysis_' date '_'];

folderNumber = 1;
folderExists = true;

% Iterate until you find a folder name that does not exist
while folderExists
    archiveFolderName = [baseFolderName sprintf('%04d', folderNumber)];
    folderExists = exist(archiveFolderName, 'dir');
    folderNumber = folderNumber + 1;
end

% Make a folder in the present working directory with that name
dos(['mkdir "' archiveFolderName '"']);

% Copy all necessary files into the directory
fprintf('\nCopying MATLAB files...\n')
dos(['copy "*.m" "' archiveFolderName '\*.m"']);
fprintf('\nCopying ANSYS Python scripts...\n')
dos(['copy "*.py" "' archiveFolderName '\*.py"']);
fprintf('\nCopying ANSYS JScript scripts...\n')
dos(['copy "*.js" "' archiveFolderName '\*.js"']);
fprintf('\nCopying Parasolid geometry files...\n')
dos(['copy "*.x_t" "' archiveFolderName '\*.x_t"']);
fprintf('\nCopying SolidWorks part files...\n')
dos(['copy "*.SLDPRT" "' archiveFolderName '\*.SLDPRT"']);
disp('All files copied.')

% Save structure 'Blade', since in optimization it may have different
% values than those in HardCodeBlade
fprintf('\nSaving Blade structure...\n')
% Write optimized value to a text file
bladeFileNumber = 1;
fileExists = true;
bladeFileName = 'BladeStructure';

% Iterate until you find a file name that does not exist
while fileExists
    fullBladeFileName = [bladeFileName sprintf('_%04d', bladeFileNumber)
'.mat'];
    fileExists = exist(fullBladeFileName, 'file');
```

```
    bladeFileNumber = bladeFileNumber + 1;
end

save([archiveFolderName '\' fullBladeFileName], 'Blade');
disp(['Blade structure saved in ' archiveFolderName '\' bladeFileName])

% Save file to pass to ANSYS archiveFolderName
fprintf('\nArchiving WorkingProject...\n')
try
    archiveFolderName_file =
fopen('E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\ArchiveD
irectory.dat', 'wt');
    fprintf(archiveFolderName_file, '%s', archiveFolderName);
% Only catch exceptions so that the file can be properly closed if there is
an error
catch err
    fclose(archiveFolderName_file);
    rethrow(err);
end
fclose(archiveFolderName_file);

% Run script for ANSYS to archive and clear Working Project
! "C:\Program Files\ANSYS Inc\v140\Framework\bin\Win64\RunWB2" -F
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\WorkingProject
.wbpj" -R
"E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\ArchiveProject
.py" -X
disp('Project archived.')

disp(['All files saved and archived in ' archiveFolderName])
```

XXIX. **ARCHIVEPROJECT.PY**

```
# Script to archive the current project to a specified folder (read in from
an outside file) and
# clear run data from the current project

# Read in the folder name
try:
    file =
open('E:\Tip_Gap_Analysis\~ProgramDevelopmentDirectory\Optimization\ArchiveDi
rectory.dat')
    archiveFolder = file.readline()
finally:
    file.close()

# Archive the project using the default settings
# Other optional parameters are specified in the Workbench Scripting PDF
documentation, page 347.
Archive(FilePath=(archiveFolder + "\ProjectArchive.wbpz"))

# Clear previous run data and messages from the project
system1 = GetSystem(Name="CFX")
component1 = system1.GetComponent(Name="Solution")
component1.Clean()
ClearMessages()

# Save the project as-is
Save()
```

XXX. WRITESPREADSHEET.M

```
% Saves data in memory from generating a speed line into an Excel file
% Intended to be run in conjunction with this directory's "Main.m"

% Save data into an Excel spreadsheet.  Save with a new filename if a
% previous name already exists
% Note: This is NOT safe to run in parallel with itself.  Doing so with the
% below file naming technique may introduce a "race condition" and result
% in both programs writing to the same file.
% Should an error cause the program to exit before this file is written,
% executing the data below this point will write the data still in memory.
% filename = 'F:\Steven_Ambers\ANSYS_Projects\Project1\Results_';
if ~exist('archiveFolderName', 'var')
    archiveFolderName = pwd;
end

number = 1;
saved = false;
disp('Saving data to spreadsheet...')
while ~saved
    excelFileName = [archiveFolderName '\Results_' sprintf('%04d', number)
'.xls'];
    if ~exist(excelFileName, 'file')
        xlswrite(excelFileName,{'Outlet Pressure [atm]'},'A1:A1');
        xlswrite(excelFileName,{'Angular Velocity [rpm]'},'B1:B1');
        xlswrite(excelFileName,{['Mass Flow, In '
outputs.mFlowInUnits]},'C1:C1');
        xlswrite(excelFileName,{['Mass Flow, Out '
outputs.mFlowOutUnits]},'D1:D1');
        xlswrite(excelFileName,{'Total-to-Total Isentropic
Efficiency'},'E1:E1');
        xlswrite(excelFileName,{['Total Pressure, Outlet '
outputs.pTotalOutUnits]},'F1:F1');
        xlswrite(excelFileName,{['Total Pressure, Inlet '
outputs.pTotalInUnits]},'G1:G1');
        xlswrite(excelFileName,{'Pressure Ratio'},'H1:H1');
        xlswrite(excelFileName,{['Power (all blades) '
outputs.powerUnits]},'I1:I1');
        xlswrite(excelFileName,pOutlet',['A2:A'    num2str(length(pOutlet)+1)]);
        xlswrite(excelFileName,omega',['B2:B' num2str(length(omega)+1)]);
        xlswrite(excelFileName,mFlowIn',['C2:C'    num2str(length(mFlowIn)+1)]);
        xlswrite(excelFileName,mFlowOut',['D2:D'
num2str(length(mFlowIn)+1)]);
        xlswrite(excelFileName,effTT',['E2:E' num2str(length(effTT)+1)]);
        xlswrite(excelFileName,pTotalOut',['F2:F'
num2str(length(pTotalOut)+1)]);
        xlswrite(excelFileName,pTotalIn',['G2:G'
num2str(length(pTotalIn)+1)]);
        xlswrite(excelFileName,pRatio',['H2:H' num2str(length(pRatio)+1)]);
        xlswrite(excelFileName,power',['I2:I' num2str(length(power)+1)]);
        saved = true;
    end
    number = number + 1;
end
```

```
disp(['Data saved in ' excelFileName]);
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Name of Addressee
   Organization of Addressee
   City, State

5. Name of Addressee
   Organization of Addressee
   City, State

Provide a separate copy of the INITIAL DISTRIBUTION LIST w/ email addresses only if you want the Research and Sponsored Programs Office (RSPO) to send the report electronically.