0.1 Algorithm Plan

0.1.1 Determinable Quantities

- Mach Number: Important because it informs the oblique shock wave angle
- Static Pressure: It is important to know the static pressure at the start of the compression cowl and the static pressure at the end of the cowl so that we can determine stagnation pressure loss using the isentropic solutions and that is an important design performance metric
- Static Temperature: Potentially useful when considering the inlet to the compressor and combustor

0.1.2 Loop Dependent Variables

- Current Step Position
- Current Step Angle
- old Step Angle theta
- New Step Angle Theta
- Turn Angle of the Ramp
- Gradient of the Ramp
- True Mach Number Before Shock
- Mach Normal to Shock Before
- Mach Nromal to Shock After
- True Mach Number After Shock
- dx

0.1.3 Needed Functionality

- Find Angle between two points
- From Angle determine gradient
- Given gradient and dl determine dx

0.1.4 Looping Procedure

1. Have an initial position in x and y
2. Have the coordinates of the cowl
3. Loop from the initial position to the specified maximum iteration count
4. Update Certain Input Variables
5. Set the Mach number before shock as mach number after shock last iteration
6. Set the old angle theta as the new angle theta from last iteration
7. Find the angle of the line between the cowl and the current position

8. update current angle
9. Compute the Mach number normal to shock
10. Use the Explicit Equation
11. update Mach Normal to Shock Before
12. Use Shock Jump relations for normal Mach number and all relevant quantities
13. Update Mach Normal to Shock After
14. Compute the turn angle of the ramp
15. Use Explicit Equation and plug in shock angle
16. update turn angle of ramp
17. Compute new true Mach number
18. Update True Mach Number After Shock
19. Find Gradient of current ramp
20. new angle theta is the old + turn angle
21. Use Basic Trigonometry to compute current gradient of ramp
22. Update Value of the gradient of ramp
23. Compute value of dx
24. Update Value of dx
25. Take one small "step" forward assuming straight line (update current position)
26. From Gradient of the Ramp and current position,
27. increment x by dx
28. increment y by gradient x dx
29. Log position and probably more quantities of interest

0.1.5 Programming Standards

- Do not use printf for printing. Let us all be consistent and use iostream and their inherited class system
- Do not edit main directly or tamper with the variable declarations. Do your individual testing in "Testing"
- Do not use "using namespace as std" instead spell out std:: this is to avoid ambiguity in namespaces

0.2 Programming Implementation

0.2.1 Header.h

A header file is declared for this simulation. All files will use the header file shown below. The header files only contain function and variable declarations. The header files are to ensure that the compiler can compile each file independently with respect to one another. The keyword `extern` is used to denote that the definition of the variable in question could be found elsewhere.

```
//Header.h
//Include Guards
#ifndef Comp_Ramp
#define Comp_Ramp

//Library Includes
#include <iostream>
//Header.h
#include <fstream>
#include <cmath>

//File Naming
#define XFileName "XRamp.txt"
#define YFileName "YRamp.txt"
#define ParamName "Parameter.txt"
#define StaticPressName "StaticPressure.txt"
#define StagPressName "StagnationPressure.txt"
#define TempName "Temperature.txt"

//Variable Prototypes
extern std::ofstream XRmpOut;
extern std::ofstream YRmpOut;
extern std::ofstream SttcPress;
extern std::ofstream StagPress;
extern std::ofstream TempOut;
extern double gam;
extern double xo;
extern double yo;
extern double xcowl;
extern double ycowl;
extern int Iter;
extern double dl;
extern double dx;
extern double xc;
extern double yc;
extern double AngCowl;
extern double Oalp;
extern double Nalp;
extern double RmpGrad;
extern double theta;
extern double M1;
extern double Mn1;
extern double M2;
extern double Mn2;
extern double P1;
extern double T1;
extern double P01;

//Function Prototypes

//Init.cpp
void Initialize();

//IO.cpp
void SetupFiles();
```

```
void CloseFiles();
void ParPrint();

//Geometry.cpp
double Pnts2Ang(double, double, double, double);
double Ang2Grad(double);
double dxComp(double);

//Gen_Rmp_Geo.cpp
void XMarch();
void GenRamp();

//Shock.cpp
double MachNormal(double, double);
double ShockMach(double);
double beta2theta(double, double);
double MachTrue(double, double, double);
double ShoCon1(double);
double ShoCon2(double);
double PAftNShock(double, double);
double TAftNShock(double, double);
double P0AftNShock(double, double);

#endif
```

0.2.2 Var_Setup.cpp

The file below is the definition of the various variables used in the program. Short comments show what the variables represent.

```
//Var_Setup.cpp
#include "Header.h"

//Flow variables
double gam;

//Initial Position
double xo; double yo;

//Cowl Position
double xcowl; double ycowl;

//Resolution of Simulation
int Iter; double dl;

//Distance in x based on dl
double dx;

//Current Position
double xc; double yc;

//Angle to Cowl
double AngCowl;

//Angle Gradients of Ramp
double Oalp; double Nalp;

//Gradient of Ramp
double RmpGrad;

//Current Turn Angle of Ramp
double theta;
```

```
//Static Pressure at entrance of diffuser
double P1;

//Static Temperature at entrance of diffuser
double T1;

//Stagnation Pressure
double P01;

//True Mach Number before Shock
double M1;

//Mach Number normal to Oblique Shock before shock
double Mn1;

//True Mach number after Shock
double M2;

//Mach Number normal to oblique shock after shock
double Mn2;
```

0.2.3 Main.cpp

The single entry point of the program is shown below. `Main` calls upon a few high level functionalities such as `GenRamp` and `SetupFiles` which will be explained later. With the grouping of basic utilities together, `Main` can be concise and show a divison in "high-level" processes occuring throughout the program,

```
//Main.cpp
#include "Header.h"

//Entry and Exit of Program
int main(void){Initialize(); SetupFiles(); ParPrint(); GenRamp(); CloseFiles(); return 0;}
```

0.2.4 Init.cpp

The file below is used to initialize the parameters in this problem. This function in truth could be modified to read the command line arguments or read from an input file.

```
#include "Header.h"

/*Computes a recurring part of the isentropic flow relation expressions*/
double IseCon(double M){return 1. + (gam-1.)/2.*pow(M,2.);}

/*Takes Static Pressure, Mach Number, and Gives Stagnation Pressure*/
double Ps2P0(double P,double M){return P*pow(IseCon(M), gam/(gam-1.));}

/*Initializes Starting Variables*/
void Initialize(){
//Ratio of Specific Heats
gam = 1.2;
//Initial Static Pressure,  Static Temperature and Mach number
P1 = 1; T1 = 298; M1 = 2.5;
//Initial stagnation pressure
P01 = Ps2P0(P1, M1);
//Initial Starting Point of Ramp
xo = 1; yo = 2;
//Location of Cowl Lip
xcowl = 5; ycowl = 3;
//Number of Iterations and what each step length is
Iter = 500; dl = 0.01;}
```

0.2.5 IO.cpp

The file handles the necessary data inputs and outputs. It is possible to use `printf` for variable output and using the `>` operator but since `C++` was used, the object-oriented method of logging data was used. Since the ramp geometry is 2-dimensional, the ramp geometry could be fully described in its x and y coordiante. For simplicity, the x-coordinate and y-coordinate of the rampare logged in 2 different files.

The parameter of the problem is printed in a separate file in case it needs to be referenced at a later time.

```
#include "Header.h"

/*Outputs X Coordinates of Ramp Geometry*/
std::ofstream XRmpOut;

/*Outputs Y Coordinates of Ramp Geometry*/
std::ofstream YRmpOut;

/*Parameter Printing*/
std::ofstream ParOut;

/*Static Pressure Printing*/
std::ofstream SttcPress;

/*Stagnation Pressure Printing*/
std::ofstream StagPress;

/*Static Temperature Printing*/
std::ofstream TempOut;

/*Prepare File Outputs*/
void SetupFiles(){
    XRmpOut.setf(std::ios_base::left, std::ios_base::adjustfield);
    XRmpOut.setf(std::ios_base::scientific, std::ios_base::floatfield);
    YRmpOut.setf(std::ios_base::left, std::ios_base::adjustfield);
    YRmpOut.setf(std::ios_base::scientific, std::ios_base::floatfield);
    SttcPress.setf(std::ios_base::scientific, std::ios_base::floatfield);
    StagPress.setf(std::ios_base::scientific, std::ios_base::floatfield);
    TempOut.setf(std::ios_base::scientific, std::ios_base::floatfield);
    XRmpOut.open(XFileName); YRmpOut.open(YFileName); ParOut.open(ParamName);
    SttcPress.open(StaticPressName);
    StagPress.open(StagPressName);
    TempOut.open(TempName);}

/*Close Files*/
void CloseFiles(){XRmpOut.close(); YRmpOut.close(); ParOut.close();}

/*Prints Problem Parameters*/
void ParPrint(){ParOut << "#gamma,␣Number␣of␣Iterations,␣length␣dl,␣Initial␣Mach␣Number,␣xcowl,␣
    ycowl"
    << std::endl; ParOut << gam << "␣" << Iter << "␣" << dl << "␣" << M1 << "␣"
    << xcowl << "␣" << ycowl;}
```

0.2.6 Gen_Rmp_Geo.cpp

The function `GenRamp` uses a single loop which indicates marching forward in distance along the ramp geometry. Note that that the Mach number and the angle of the ramp geometry are sufficient parameters to characterize position on the ramp. All quantities can be derived from just those 2 parameters. Knowing that, the function `XMarch` overwrites the 2 parameters so that the numerical simulation can "march" in space.

```
//Gen_Rmp_Geo.cpp
#include "Header.h"

/*Overwrites last position variables with new position*/
void XMarch(){M1 = M2; Oalp = Nalp;}

/*Generates Ramp Geometry*/
void GenRamp(){for(int i=0; i<Iter; i++){
    AngCowl = Pnts2Ang(xc,yc,xcowl,ycowl);
    Mn1 = MachNormal(M1, AngCowl);
    //Function calls to Static Pressure
    P1 = PAftNShock(Mn1, P1);
    //Function calls to stagnation pressure
    P01 = P0AftNShock(Mn1, P01);
    //function calls to static temperature
    T1 = TAftNShock(Mn1, T1);
    Mn2 = ShockMach(Mn1);
    theta = beta2theta(M1,AngCowl);
    M2 = MachTrue(Mn2,AngCowl,theta);
    Nalp = Oalp + theta;
    RmpGrad = Ang2Grad(Nalp);
    dx = dxComp(RmpGrad);
    xc = xc+dx; yc = yc+RmpGrad*dx;
    XRmpOut << xc << "␣"; YRmpOut << yc << "␣";
    SttcPress << P1 << "␣"; StagPress << P01 << "␣";
    TempOut << T1 << "␣";
    XMarch();}}
```

0.2.7 Geometry.cpp

The functions implemented below are purely kinematic. They are based on basic trigonometry and basic algebraic manipulations of differential distances. The function `Pnts2Ang` is used when determining the wave angle of the oblique shock meanwhile the functions `Ang2Grad` and `dxComp` are used to determine the step in distance x such that the length of the ramp dl is preserved.

```
//Geometry.cpp
#include "Header.h"

/*Takes 2 points and returns the angle between them*/
double Pnts2Ang(double x1, double y1, double x2, double y2){return atan((y2-y1)/(x2-x1));}

/*Takes an angle a line makes with the x-axis and returns the gradient of that line*/
double Ang2Grad(double angle){return tan(angle);}

/*Using a gradient to determine dx*/
double dxComp(double gradient){return dl/sqrt(1. + pow(gradient,2));}
```

0.2.8 Shock.cpp

The file below is the programming implementation of the various isentropic flow relations.

```
//Shock.cpp
#include "Header.h"

/*Computes the Normal Component of Mach number on an Oblique Shock*/
double MachNormal(double Mach, double beta){return Mach*sin(beta);}

/*Computes Mach Number after Normal Shock*/
double ShockMach(double Mach){
    return sqrt((1.+(gam-1.)/2.*pow(Mach,2))/(gam*pow(Mach,2)-(gam-1.)/2.));}
```

```
/*Computes turn angle theta from shock angle beta*/
double beta2theta(double Mach, double beta){
    return atan(2./tan(beta)*(pow(Mach,2)*pow(sin(beta),2)-1.)/(pow(Mach,2)*(gam+cos(2*beta))+2.));}

/*Computes the True Mach Number from Normal Mach number, shock angle and turn angle*/
double MachTrue(double NMach, double beta, double theta){
    return NMach/sin(beta-theta);}

/*First Commonly Recurring Constant in Shocks*/
double ShoCon1(double Mach) { return (1 + (2. * gam) / (gam + 1) * (pow(Mach, 2.) - 1)); }

/*Second Commonly Recurring Constant in Shocks*/
double ShoCon2(double Mach) { return ((2 + (gam - 1) * pow(Mach, 2)) / ((gam + 1) * pow(Mach, 2)));
    }

/*Computes the Static Pressure after a normal Shock Using Static Pressure*/
double PAftNShock(double Mach, double Pressure) {
    return Pressure * (1. + (2. * gam) / (gam + 1.) * (pow(Mach, 2.) - 1.));}

/*Computes the Static Temperature afer a Normal Shock using static temperature before shock*/
double TAftNShock(double Mach, double Temperature) {
    return Temperature * ShoCon1(Mach) * ShoCon2(Mach);}

/*Computes the Stagnation Pressure After a Normal Shock using Stagnation Pressure Before shock*/
double P0AftNShock(double Mach, double Pressure){
    return Pressure * pow((ShoCon1(Mach)), (-1. / (gam - 1.))) * pow((ShoCon2(Mach))
        ,(-gam / (gam - 1.)));}
```

0.3 Build Tools

The `Makefile` utility is used to build the binary of this project. The various functions are compiled independently of each other into object files. Then, they are compiled together to form the final binary. The script that automates this process is shown below. More complex cross-platform build tools might be available such as `CMake`. It seems that this project may be buildable on a unix-like system for now.

```
#List of all object files
Files = Gen_Rmp_Geo.o Geometry.o IO.o Init.o Main.o Shock.o Var_Setup.o

#Compiler Cho.cppe
Compiler = g++

#Compile Flags
Flags = -lm

#Final Binary
Exec: $(Files)
  @echo "Linking␣all␣Object␣Files"
  @$(Compiler) $(Flags) $(Files) -o Exec

Gen_Rmp_Geo.o: Gen_Rmp_Geo.cpp
  @echo "Building␣Gen_Rmp_Geo..."
  @$(Compiler) $(Flags) -o Gen_Rmp_Geo.o -c Gen_Rmp_Geo.cpp

Geometry.o: Geometry.cpp
  @echo "Building␣Geometry..."
  @$(Compiler) $(Flags) -o Geometry.o -c Geometry.cpp

IO.o: IO.cpp
  @echo "Building␣IO..."
  @$(Compiler) $(Flags) -o IO.o -c IO.cpp

Init.o: Init.cpp
  @echo "Building␣Init..."
  @$(Compiler) $(Flags) -o Init.o -c Init.cpp

Main.o: Main.cpp
  @echo "Building␣Main..."
  @$(Compiler) $(Flags) -o Main.o -c Main.cpp

Shock.o: Shock.cpp
  @echo "Building␣Shock..."
  @$(Compiler) $(Flags) -o Shock.o -c Shock.cpp

Var_Setup.o: Var_Setup.cpp
  @echo "Building␣Var_Setup..."
  @$(Compiler) $(Flags) -o Var_Setup.o -c Var_Setup.cpp

#sub.o: sub.cpp
# @echo "Building sub..."
# @$(Compiler) $(Flags) -o sub.o -c sub.cpp

clean:
  @echo "Removing␣Build␣Relics"
  @rm *.o
```

0.4 Post-Processing Utilities

Visualization of the ramp geometry is done in python. The python script that shows the ramp geometry is shown below,

```
import numpy as np
import matplotlib.pyplot as plt

#Load Problem Parameter
param = np.genfromtxt('Parameter.txt')

#Load Data
xramp = np.genfromtxt('XRamp.txt')
yramp = np.genfromtxt('YRamp.txt')
sttcpress = np.genfromtxt('StaticPressure.txt')
stagpress = np.genfromtxt('StagnationPressure.txt')
tempout = np.genfromtxt('Temperature.txt')

#Basic Plot Command
figure, axis = plt.subplots(2,2)

#Plot All Quantities
axis[0, 0].plot(xramp,yramp, '-k', label='Ramp␣Geometry')
axis[0, 0].plot(param[4], param[5], 'rx', label = 'Cowl')
axis[1, 0].plot(xramp,sttcpress, '-k', label='Static␣Pressure')
axis[1, 1].plot(xramp,stagpress, '-k', label='Stagnation␣Pressure')
axis[0, 1].plot(xramp,tempout, '-k', label='Temperature')

#Ramp Plot Settings
axis[0, 0].set_xlabel('x-coordinate␣(m)')
axis[0, 0].set_ylabel('y-coordinate␣(m)')
axis[0, 0].set_title('Ramp␣Geometry')
axis[0, 0].set_aspect('equal')
axis[0, 0].legend()
axis[0, 0].grid()

#Static Pressure Plot Settings
axis[1, 0].set_xlabel('x-coordinate␣(m)')
axis[1, 0].set_ylabel('Pressure␣(kPa)')
axis[1, 0].set_title('Static␣Pressure')
#axis[1, 0].set_aspect('equal')
axis[1, 0].legend()
axis[1, 0].grid()

#Stagnation Pressure Plot Settings
axis[1, 1].set_xlabel('x-coordinate␣(m)')
axis[1, 1].set_ylabel('Pressure␣(kPa)')
axis[1, 1].set_title('Stagnation␣Pressure')
#axis[1, 1].set_aspect('equal')
axis[1, 1].legend()
axis[1, 1].grid()

#Static Temperature Plot Settings
axis[0, 1].set_xlabel('x-coordinate␣(m)')
axis[0, 1].set_ylabel('Temperature␣(K)')
axis[0, 1].set_title('Temperature')
#axis[0, 1].set_aspect('equal')
axis[0, 1].legend()
axis[0, 1].grid()

#Show Plot Command
plt.show()
```