# Lowrider to Flowrider
# Vehicle Aerodynamics and Aeromodding

New Mexico

Supercomputing Challenge

Final Report

April 3, 2013

Team 53

Los Alamos High School

**Team Members**
Rachel Robey
Simon Redman

**Teacher**
Lee Goodwin

**Project Mentors**
Robert Robey
Jim Redman

**Abstract**

The goal of this project is to investigate the improvement of gas mileage by reducing the air drag on vehicles. Rather than designing body-up, we focus on the improvement of existing cars. To that end, we seek to create a set of tools to identify the parts of vehicles that are most advantageous to target with aeromodding modifications and to quickly determine the impact of proposed modifications. We regard this as a work in progress and have approached the problem from both a modeling perspective and an experimental one. We currently have a means of export from a CAD program, a module to generate a computational mesh for the given 2D polygon object, and an incompressible CFD model on a regular grid. We have begun work on developing and implementing the cut-cell version of the CFD, but it has not yet been completed. As such, we have not yet reached a stage where it is possible to produce relevant results, but would like to validate it against experimental data if possible. We will be using an electric car to test, as it is much easier to measure electrical power than then gasoline usage. All the instrumentation is in place and has been tested but has not yet been used to collect data.

## 1 Introduction

Computer aerodynamic modeling is nothing new. The auto industry uses computer models of their cars to help improve their aerodynamics. However, there are no aerodynamic models available to the average consumer. Normally this would not be a problem, as most consumers do not need to know and rarely even care how aerodynamic their car is. There are some people, though, who make aftermarket aerodynamic improvements to their car, which is known as aeromodding. For these people, knowing how effective a change is going to be without first having to put it on the car would be very helpful.

Rebuilding a new car to be more efficient is much more environmentally friendly than buying a new, more efficient, car. According to Wired[8], a new second-generation Prius has already consumed the equivalent energy cost of 1000 gallons of gasoline before it even reaches the road. Rebuilding avoids the costs of making a new car, and it avoids the costs of having to dispose of the old car. In general, there are two main possibilities for rebuilding a car.

First, many people take an old car with a non-working engine and convert it to a more efficient fuel type, such as electric. Unfortunately, the downsides of alternative fuels, such as low range and lack of infrastructure, are usually more pronounced in aftermarket conversions because often old cars were not designed with efficiency in mind and the converter will only put in exactly as many batteries as he needs to keep costs down. Therefore, many people who convert cars are also interested in the second possibility for rebuilding.

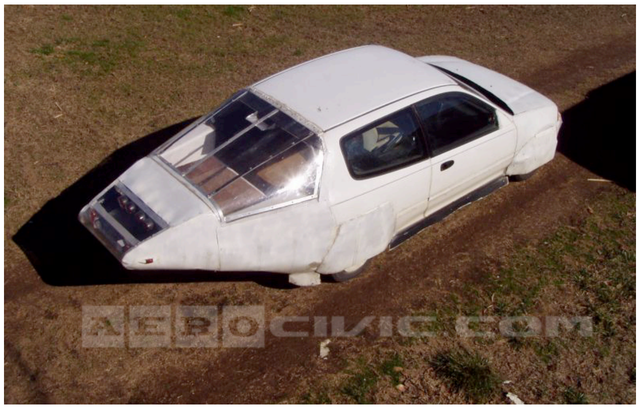

Figure 1: The Aerocivic, an example of an extensive aeromod (picture credit [9])

If the car still runs, and the person simply wants to improve its fuel efficiency, aeromodding is often a good alternate course of action. Aeromodding means taking the vehicle and making it more aerodynamic. The possibilities for aeromodding range from putting on new, smoother hubcaps, to putting a cover over the entire wheel well, to redesigning the whole body of the car. Obviously, the more extensive the modification is, the greater the potential for lowering the coefficient of drag on the vehicle.

The Aerocivic is a well-known success story in the aeromoddering community[9]. After applying many popular techniques, such as adding a boat tail and moving the wing mirrors inside, the coefficient of drag dropped from 0.34 to 0.17. The Aerocivic gets about 70 miles per gallon, two-thirds better gas mileage than a stock Honda Civic. The Aerocivic demonstrates that aerodynamic modifications can have a significant improvement on car gas mileage. It is important, therefore, to encourage these types of projects.

## 2 Problem Statement

While it is possible to test proposed car modifications by actually building them and applying them to the car, using a modern computer to test a modification is much faster and less costly. Allowing aeromodders to quickly tests many different ideas will allow them to come up with a more valuable solution and visualize the results of their change. Our goal is to create a program which calculates the air drag of a modeled car and allows the model to easily be changed to simulate proposed modifications. In order to validate our model, we will compare the model's results with the results we get from testing a real-world vehicle. Our test vehicle is electric because it is very easy to measure the energy usage of an electric car.

## 3 Physical Model

Once we have a working model, it is important to be able to prove that it produces realistic results. To do this, we will compare the results gained from the model with measurements taken from real-world tests.

### 3.1 Test Vehicle

Figure 2: An electric Suzuki Sidekick. This is the car we will use for real-world tests.

The test vehicle selected as the physical model is an aftermarket converted electric Suzuki Sidekick. The Sidekick is a small, four-wheel drive SUV which is powered by a huge forklift motor connected to the original transmission through the original clutch. The battery is 32 3.2 volt 10 amp-hour lithium Li-PO batteries which provide an estimated range of about 30 miles.

### 3.2 Measuring Changes in Energy Use

There are two tests that could be used to determine the effectiveness of a modification. The first is called the Rolldown Test, and is described on the Instructables page by iwilltry[5]. It involves getting the car up to speed and measuring how long it takes to slow down while coasting on a flat road. To improve the data, the test is usually done going both ways along the same piece of road, to accommodate for any road grade, and the data from several runs is averaged to reduce error. This test is very popular because it doesn't rely on any special instrumentation being installed in the car and it is straight-forward to execute.

The second test gets data on the car's efficiency by directly measuring the car's energy usage. The test can be repeated after a modification along the same or similar piece of road to get an idea of the effectiveness of each modifications. This test is less popular because it requires some kind of measuring device to be installed in the car. However, it gives data which can easily be used to show how well a modification worked at a certain speed. In a gas car, the instrumentation is usually something which measures the amount of gas leaving the gas tank, meaning that it has to be installed somewhere on the gas line. In an electric car, the idea is the same. Some system must be put in place to measure how much energy is leaving the battery. However, unlike gas cars, many electric cars already have a shunt, which is effectively a high-power, precisely calibrated resistor, and an ammeter in place since it is helpful to know how much power is being used in order to avoid damaging the battery cells.

Because of the advantage of being able to measure the energy usage at any speed and because our test car is electric, we have decided to use the second test to collect data to compare with the model.

### 3.3 Conducting Tests

The electric Sidekick already has a shunt in the main wire that comes from the battery to the motor. However, rather than use a meter and a person taking readings every few seconds, we have decided to use an Arduino with bluetooth capabilities to collect and output the data. With this set up, we are able to take running readings at practically any time interval. The data is then output to a device reading the bluetooth comm, such as an Android device or a laptop, which logs the data for later viewing.

The Arduino is connected to the car's main shunt using a differential op-amp circuit, which is a circuit used to amplify a voltage, to get the tiny voltage on the shunt into a number readable by the Arduino. Once we know the voltage on the shunt, we can calculate the amperage and the power.

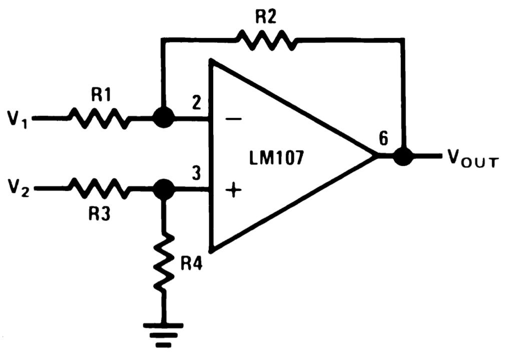

Figure 3: A differential op-amp circuit. The values of the resistors determine the magnitude of the amplification (image credit [11]).

## 4 Computational Methodology

The model thus far consists of two general components: geometry and fluid flow. For the sake of project scale, we have chosen to keep the computational fluid dynamics (CFD) model fairly simple in order to focus more on the geometry aspect. The model is still a work in progress and several of the parts have been developed as a separate modules that have not yet been integrated. We present the work thus far, an overview of how it fits into the model as a whole, and some of the theory in as yet unimplemented parts of the model.

### 4.1 Geometry

#### 4.1.1 Inputting Object with CAD Program

There are many existing programs to facilitate the design of geometrically complex objects. The first step of the process of resolving an object in a computational model is to generate the necessary geometric information. As far as streamlining and easing the process of inputting the geometric information, computer-aided design (CAD) programs have become fairly popular and sophisticated.

We did some work with Google Sketch-Up, a free 3D-modeling program, as an existing option with an easy-to-use interface. Only a limited amount of information would actually need to be exported for our use and a generic file format was not necessary. This made using Sketch-Up's ruby application programming interface (API) an ideal choice[7]. Through an imported ruby script, the vertices and faces of an object designed in Sketch-Up can be readily written to a file and retrieved in the next steps in the process.

This script has been successfully written and implemented, but is separate from the rest of the model at this point. We are only working in two dimensions and while it is possible to design and export on a plane, the development of the model has not progressed to the point that we are importing complex objects or polygons. While it has not been used as such, we consider this to be a viable and very feasible way to import the geometry of objects and will be an important step in later stages of the project.

### 4.1.2 Resolving Geometry on a Computational Mesh

The description of the object (in two dimensions, a polygon) as a set of vertices must be transformed into usable geometric information in the computational model. We have chosen to resolve flow around the object through a cut-cell method, which is described in more detail in Section 4.3. At this point, it is only relevant insofar as the data that needs to be output.

Given the object of interest, a mesh is generated to be overlaid on it. Each cell should then be clipped along the polygon edges. From these cut cells, a few geometric properties are required for the cut-cell computational model. In particular.

- $\alpha_{i,j}$ - the volume of the cut cell, relative to that of a regular cell
- $\beta_{i\pm\frac{1}{2},j}$ and $\beta_{i,j\pm\frac{1}{2}}$ - the area of the interface with cells in each direction (left, right, bottom, top) relative to that of a regular cell

These quantities will be used to make the adjustments to volumen and flux in the computational model.

**Polygon Clipping** In order to generate the clipped cells and find the relevant data to export, we turn to graphic algorithms. Graphics tend to be strong when working with geometry and space and have already developed algorithms for many of these operations.

Based on extreme coordinates (furthest left/right/down/up) a mesh of given cell size is overlaid on the polygon describing the object in consideration. Thus arranged, each computational cell is regarded as a polygon to be clipped along the edges of the object polygon. We implement a modified version of the Sutherland/Hodgman algorithm[4].

The original algorithm, having been developed for graphics applications, presents the clipping of polygons against planes of a viewport. Polygons within a volume or viewing plane are repeatedly clipped along the various planes of the viewport to remove the parts that lie outside of view. Our application has required some modifications as it requires clipping *around* a polygon rather than inside *one*. In addition to describing the primary algorithm, we also address the alternate issues in that arose.

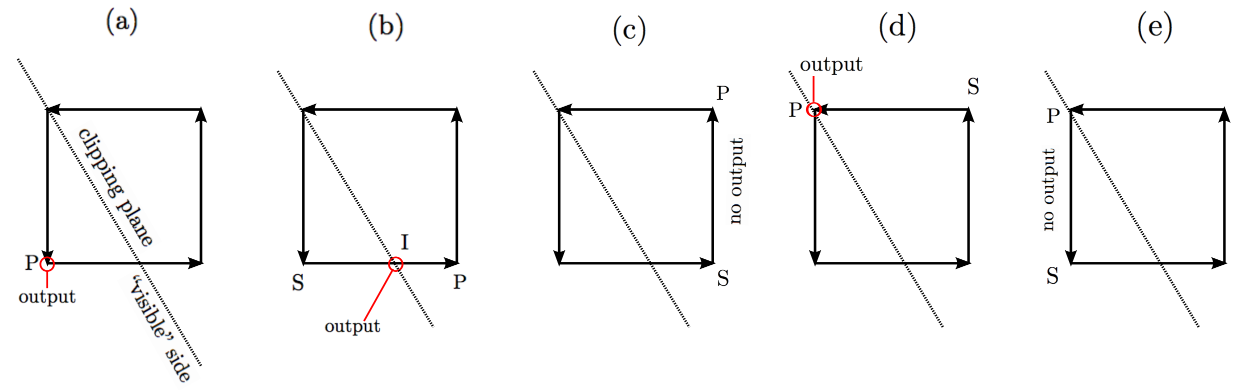

Figure 4: Example of polygon clipping applied to a cell: (a) the first point is 'visible' and therefore output (b) the line crosses the clipping plane and the intersection point is also output, while point $P$ is not (c) both points are on the 'invisible' side of the plane and nothing is added (d) the end point is considered 'visible' and added to the array (e) the segment does not intersect the plane and the final point is not included. The collection of output points describe the cut cell.

The procedure for clipping a polygon against a single plane, such as an edge of the object polygon, is described by Sutherland and Hodgman[4]. An ordered set of vertices describes each cell: $P_1, \ldots, P_n$. Excepting the initial vertex, each point is considered as the terminal end of an edge; that is, the edge defined by the current vertex and the previously input one. The points must be tested for 'visibility', meaning they are on the side of the plane that is not being removed, while the edge must be tested for intersection with the plane. The result is a new ordered set of vertices which describe the clipped polygon.

The initial vertex is tested for 'visibility'. If positive, and the cell appears on the retained side of the plane, it is added as the first of the output vertices. Otherwise, the algorithm continues without doing anything. The process continues with the rest of the vertices in order; each vertex $P$ is considered with the proceeding vertex $S$. If the line segment $\overline{PS}$ crosses the clipping plane, the intersection point is computed and output. Then $P$ itself is considered for 'visibility' and a 'visible' point is output. These output points describe the clipped version of the polygon. Figure 4 illustrates this process as applied to a sample cell where an edge of the object constitutes the clipping plane.

The mesh can then be completely clipped to the embedded object by successively clipping the cells along each of the edges. Some additional fine-tuning adjustment must be made, however. When clipping a side, the cells along the segment are singled out to be clipped through simple interpolation. To handle the case of an edge landing mid-cell, the clipping planes must be modified into line segments. The test for intersection treats the line as infinite and will clip beyond the edge of the object polygon. Therefore, an additional test is added to check if computed intersection points lie within the range of the line segment. A further issue arises when vertices are prematurely removed. In clipping one edge, a vertex within the object may be removed which would otherwise generate an intersection point when the next edge was clipped. However, since it has already been removed, the additional side is lost. We account for the problem by passing not two but three vertices during a clip. Adding the next vertex effectively lets not only the current edge, but also the following edge to be considered. Only the cells along the current edge are clipped, so this measure affects primarily the cells about the corners of the object polygon. While the above modifications counter the main problems that have arisen in our adaptation of the polygon clipping algorithm, some complications still remain. The main failure we have seen is on sharp corners, in which a cell straddles the polygon and the vertices on either side register as being 'visible', causing that portion of the object to be overlooked. These cases could be separately accounted for, but as the problem can be lessened through increased resolution and as most car shapes have larger, more obtuse angles that are handled correctly, it does not significantly affect our particular application.

Figure 5 shows visualizations of the clipped mesh about various embedded polygons. The truncation of corners can be clearly seen on those with internal angles of 90° or less when they align with the cells. The truncation at smaller resolution occurs further down the corner and would presumably have a smaller impact. Other inconsistencies between the original polygon and clipped mesh still exist, but the overall representation is fairly accurate.

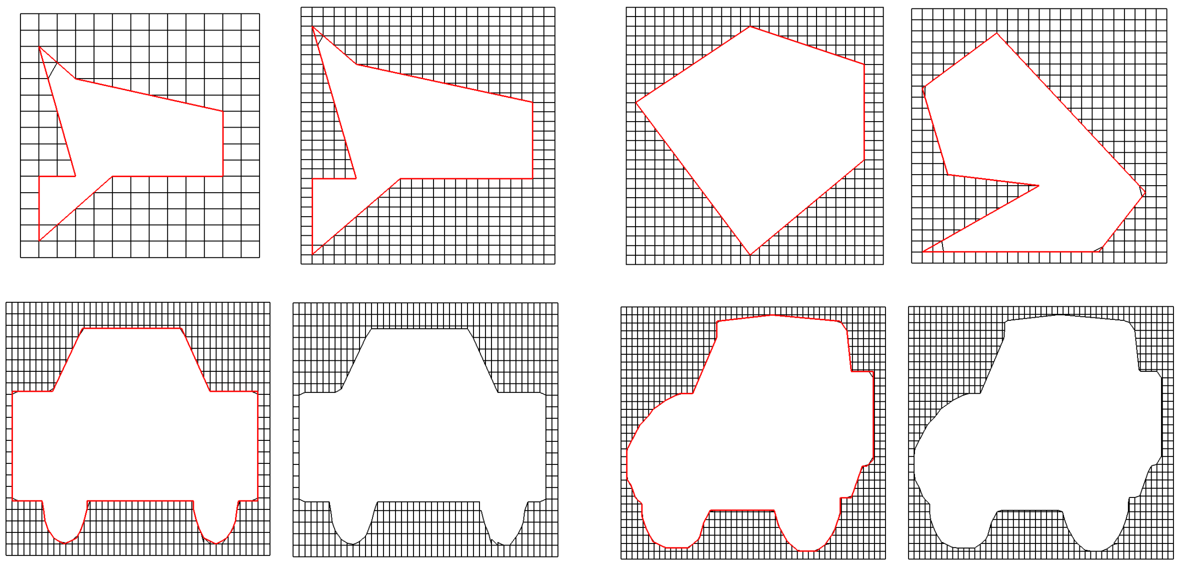

Figure 5: Visualization of clipping applied to several different embedded polygons (written in OpenGL). The first set shows the same shape at different resolutions and the top right shows more simple shapes. The bottom row shows two different car-like shapes with and without the original polygon outline. Stretched cells result from display proportions.

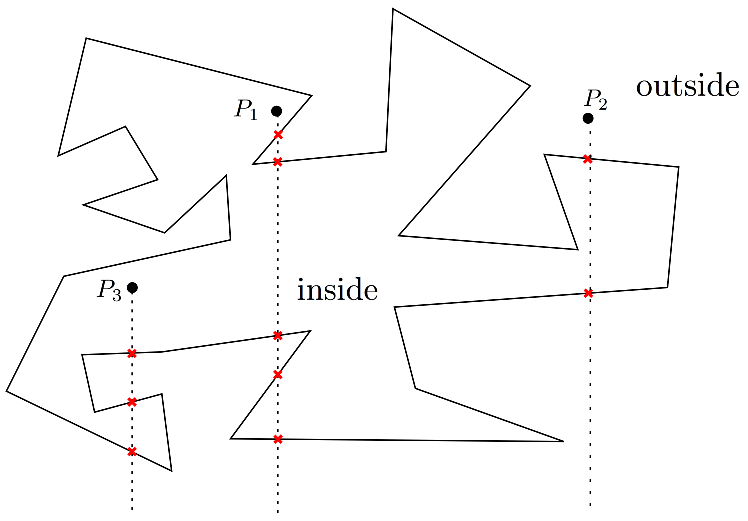

Figure 6: Illustration of Jordan Curve Theorem and crossings of polygon on a ray in the $-y$ direction

**Point-In-Polygon Test** Rather than being determined solely by which side of a plane a point lies on, 'visibility' in our application is determined by whether the point lies inside or outside the object polygon. Points outside the object are relevant parts of the mesh, while those inside lie within the rigid object and are removed. Point-in-polygon tests are common in graphics. The one used in our model is based on the concept Jordan Curve Theorem and implemented with the help of an online description[10].

The Jordan Curve Theorem states that a polygon (or any closed body) divides the plane into two distinct parts: an inside and an outside. Crossing any portion of the polygon alternates from one to the other (see Figure 6).

Using this fact, an arbitrary ray can be drawn from any given test point and the number of crossings of the ray counted. Since the infinitely extending ray extends far beyond the polygon, it ultimately reaches the outside portion of the plane. Counting backwards, an even number of crossings indicates the test point lies outside the polygon while an odd count signifies an inside point. For convenience, a simple downward ray is used.

Vertices present a special case. The ray may be either just brushing the vertex and not crossing the polygon or simply crossing the polygon at the vertex. A negligible nudge to either side addresses the issue. A touched vertex will then be resolved into either no crossings or one crossing after another. An actual crossing has an edge on either side from the perspective of the test point and a crossing will be counted.

The way in which the test is implemented enables points on the polygon boundary to be separately identified. An additional argument is passed to the routine to designate the desired return value in such a case, thereby allowing 'outside' to be returned when determining 'visibility' and 'inside' when testing whether cells are completely contained.

### 4.1.3 Relevant Equations

Here is a quick note as to some of the relevant equations used to implement the algorithm described above.

The method of determining crossings of the clipping plane and computing the intersection point utilizes a convenient distance measure from end points to plane[4]. The segments need only be parallel and as such, are measured parallel to the axis. Crossing of the plane (i.e. the end points lay on opposite sides of the clipping plane) is easily distinguished by opposite-signed distance measures. Otherwise proportions and similar triangles are applied to find the intersection point:

$$I = \gamma P_2 + (1-\gamma) P_1 - P_1 + \gamma (P_2 - P_1)$$

with endpoints $P_1$ and $P_2$ and $\gamma$ denoting the fraction of the line $\overline{P_1P_2}$ before the intersection point. With the distance measures $\overline{P_1R_1}$ and $\overline{P_2R_2}$, these conditions for the expression above are given as:

$$\frac{|P_1R_1|}{|P_2R_2|} = \frac{|P_1I|}{|P_1I|}$$

$$\gamma = \frac{|P_1I|}{|P_1P_2|} = \frac{|P_1R_1|}{|P_1R_1 + |P_2R_2||}$$

Additionally, in the geometry export, the area must be determined from the polygon vertices. The area of the polygon described by points $(x_1, y_1), \cdots (x_n, y_n)$ is given by

$$Area = \left| \frac{(x_1y_2 - y_1x_2) + (x_2y_3 - y_2x_3) + \cdots + (x_ny_1 - y_nx_1)}{2} \right|$$

There are limitations for self-intersecting polygons, but these should not appear in the clipped cells.

## 4.2 Fluid Dynamics

There are many sophisticated fluid dynamics models, however we take one of the most basic and straightforward schemes, the rationale being to ease the merging of the geometric side of the project and save time. In the absence of shocks or other phenomena that cause significant compression, incompressible fluid flow should adequately represent normal air about a car body. The regular-mesh incompressible CFD was implemented as outlined by Harlow and Scannapieco[3] and is a staggered Eulerian model with cell-centered interpolations

An incompressible model assumes constant density and no changes in energy due to work, thereby reducing the number of factors that must be addressed. The main variables are pressure $P_{i,j}$, horizontal velocity $u_{i+\frac{1}{2},j}$, and vertical velocity $v_{i,j+\frac{1}{2}}$. As implied by their indices, the physical location of the stored values of these variables on a staggered grid follows the graphical representation in Figure 7.

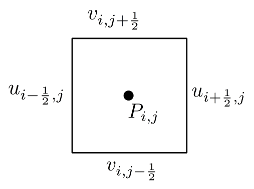

Figure 7: Variables of pressure $P$ and velocities $u_{i\pm\frac{1}{2},j}$ and $v_{i,j\pm\frac{1}{2}}$ placed in their respective locations on a staggered mesh.

The transport equation in two dimensions, which represents the advective flow of mass, is shown below.

$$\frac{\delta \rho}{\delta t} + \frac{\delta \rho u}{\delta x} + \frac{\delta \rho v}{\delta y} = 0 \implies \frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} = 0$$

The density ($\rho$), may be dropped because is assumed to remain constant. The next consideration is of momentum, which is solved for momentum cells staggered to lie centered on locations where velocity is directly represented (see Figure 8). Considering and condensing the flux over each interface of the momentum cell, the change in momentum due to advection assumes the term

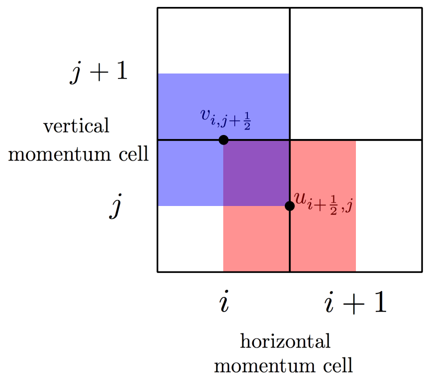

Figure 8: Location of momentum cells on staggered mesh

$$\left(\frac{\delta \rho u}{\delta t}\right)_{adv} = -\frac{\delta \rho u^2}{\delta x} - \frac{\delta \rho uv}{\delta y}$$

Lastly, pressure results from the combination of real pressure and viscous pressure. Summed, they contribute the term

$$\frac{\delta p}{\delta x} - \rho \nu \left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right)$$

with $\nu$ as the kinematic viscosity.

The combination of these terms gives equations for the change in momentum in the x direction and the similar equation for the y direction.

$$\frac{\delta u}{\delta t} + \frac{\delta u^2}{\delta x} + \frac{\delta uv}{\delta y} = -\frac{\delta P}{\delta x} + \nu \left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right)$$

and

$$\frac{\delta v}{\delta t} + \frac{\delta uv}{\delta x} + \frac{\delta v^2}{\delta y} = -\frac{\delta P}{\delta x} - \nu\left(\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2}\right)$$

In solving these equations, pressure is separated out from the other terms to be solved implicitly once the advective and viscous terms have been found explicitly. Excluding contributions made by pressure, the discretized versions of the updates are given as

$$\begin{aligned}\bar{u}^{n+1}_{i+\frac{1}{2},j} &= u^n_{i+\frac{1}{2},j} - dt\Bigg[\left(\frac{u^2_{i+1,j} - u^2_{i,j}}{dx}\right) + \left(\frac{(uv)_{i+\frac{1}{2},j+\frac{1}{2}} - (uv)_{i+\frac{1}{2},j-\frac{1}{2}}}{dy}\right) \\ &\quad -\nu\left(\frac{u_{i+\frac{3}{2},j} + u_{i-\frac{1}{2},j} - 2u_{i+\frac{1}{2},j}}{dx^2}\right) - \nu\left(\frac{u_{i+\frac{1}{2},j+1} + u_{i+\frac{1}{2},j-1} - 2u_{i+\frac{1}{2},j}}{dy^2}\right)\Bigg]\end{aligned}$$

$$\begin{aligned}\bar{v}^{n+1}_{i,j+\frac{1}{2}} &= v^n_{i,j+\frac{1}{2}} - dt\Bigg[\left(\frac{(uv)_{i+\frac{1}{2},j+\frac{1}{2}} - (uv)_{i-\frac{1}{2},j+\frac{1}{2}}}{dx}\right) + \left(\frac{v^2_{i,j+1} - v^2_{i,j}}{dx}\right) \\ &\quad -\nu\left(\frac{v_{i+1,j+\frac{1}{2}} + v_{i+1,j+\frac{1}{2}} - 2v_{i,j+\frac{1}{2}}}{dx^2}\right) - \nu\left(\frac{v_{i,j+\frac{3}{2}} + v_{i,j-\frac{1}{2}} - 2v_{i,j+\frac{1}{2}}}{dy^2}\right)\Bigg]\end{aligned}$$

Pressure can then be found by minimizing the overall change in mass

$$D_{i,j} = \frac{u_{i-\frac{1}{2},j} - u_{i+\frac{1}{2},j}}{dx} + \frac{v_{i,j-\frac{1}{2}} - v_{i,j+\frac{1}{2}}}{dy}$$

to zero with Newton's Method.

With this regular-grid model, it is possible to model a limited variety objects in the flow. Whole interfaces can be set to have no flow and an object described by the blocked off cells. Figure 9 shows the flow about some rectangles on the boundary and in the middle of the domain.

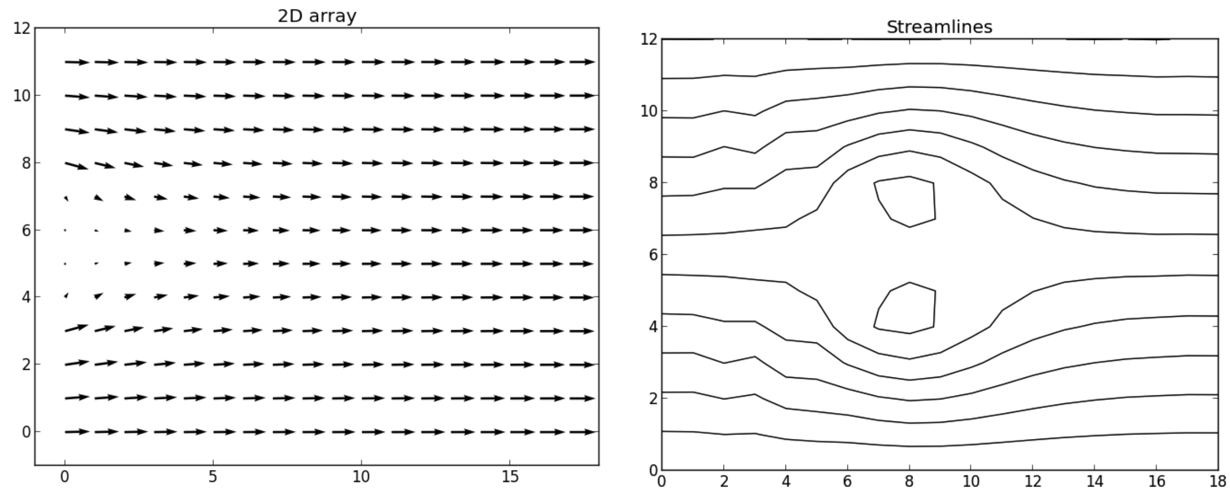

Figure 9: Flow about rectilinear obstacles in a flow from the regular grid CFD and visualized as (left) vector field and (right) streamlines.

## 4.3 Cut-Cell Modifications

We are using a cut-cell (also called embedded or immersed boundary) to deal with the non-rectilinear shapes of the obstacle objects. The regular computational model must be adjusted to cope with partial cells. The basic concept behind the method used is based primarily off of the paper *Well-balanced compressible cut-cell simulation of atmospheric flow* by Klein, Bates, and Nikiforakis[4] with modifications made and derived from the ideas presented in the paper.

The original intention was to integrate the fluid dynamics equations – developed on the regular grid and described above – with the cut-cell methodology. However, this requires a clear description of flux at cell boundaries that is muddled by the staggered grid. A cell-centered arrangement would likely ease development. At this point, we present the cut-cell modifications in terms of cell interface fluxes, omitting for the time being the application to the CFD equations.

The cut-cell discussion begins with the 1-dimensional version as the extension to two dimensions heavily depends upon it.

### 4.3.1 One-Dimensional Cut-Cells

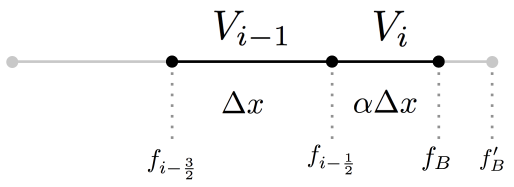

Figure 10: One-dimensional cut-cell problem; the partial cell lies to the right of the regular cell and has a length of fraction $\alpha$ to a normal cell length.

The evolvement of conserved state variables is described by a general form of a standard explicit conservative update:

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n - \frac{\Delta t}{\Delta x}\left(\mathbf{f}_{i+\frac{1}{2}} - \mathbf{f}_{i-\frac{1}{2}}\right)$$

in which $\mathrm{u}_i^n$ represents the vector of conserved state variables at time $n$ in a cell of index $i$. $\mathbf{f}_{i\pm\frac{1}{2}}$ denotes the numerical flux at either boundary of the cell. A major concern for cut-cell methods is maintaining stability at a reasonable time step despite the relatively small fragments of cells that may be left after clipping. In one dimension, a solution can be to weight the flux, thereby 'extending its influence'. A stabilized flux, referring to Figure 10, is

$$f^*_{i-\frac{1}{2}} = f_B + \alpha_i\left(f_{i-\frac{1}{2}} - f_B\right)$$

### 4.3.2 Two-Dimensional Cut-Cells

The two-dimensional problem can be solved by reducing it into a combination of one-dimensional problems. This is complicated by the fact that embedded boundaries do not necessarily conform to the coordinate directions.

The two-dimensional cell is first decomposed into two parts: a shielded and an unshielded portion. The first corresponds to the part blocked by the boundary in the current direction. The second acts like a narrower, unblocked cell. The total flux collects the flux terms weighted according to the interface areas.

$$\mathbf{f}^{2D}_{i-\frac{1}{2},j} = \frac{1}{\beta_{i+\frac{1}{2},j}}\left[\beta_{i+\frac{1}{2},j}\mathbf{f}^{unshielded}_{i-\frac{1}{2},j} + \left(\beta_{i-\frac{1}{2},j} - \beta_{i+\frac{1}{2},j}\right)\mathbf{f}^{shielded}_{i+\frac{1}{2},j}\right]$$

The unshielded flux is unaffected by the cut cell except for the area weighting applied in the above equation. It is computed normally:

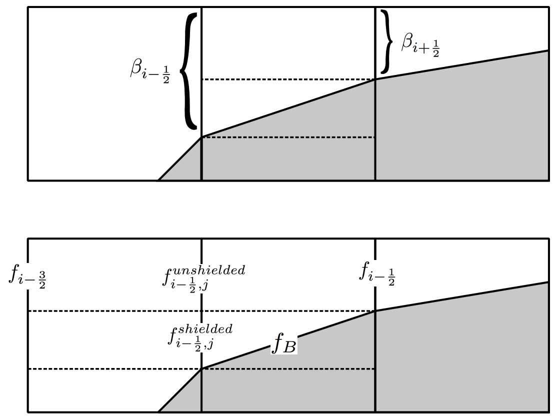

Figure 11: Decomposition of two-dimensional cut-cell

$$\mathbf{f}^{unshielded}_{i-\frac{1}{2},j} = \mathbf{f}_{i-\frac{1}{2},j}$$

The shielded portion, however, is solved as a one-dimensional cut cell problem as presented above. The effective length of the cell being taken as the average distance to the embedded boundary. The ratio seen in the one-dimensional sample can be found from the relative volume and interfaces as

$$\alpha^{shielded}_{i-\frac{1}{2},j} = \frac{\alpha_{i,j} - \beta_{i+\frac{1}{2},j}}{\beta_{i-\frac{1}{2},j} - \beta_{i+\frac{1}{2},j}}$$

as given by Klein, et al. [6]. Here we show our own brief derivation of the expression.

The shielded volume is given by $\alpha_{i,j} - \left(\beta_{i+\frac{1}{2},j}\right)(1)$ where 1 denotes 100% of the normal top interface. This volume is set equal to the sum of the ...

$$\alpha_{i,j} - \beta_{i+\frac{1}{2},j} = x_2\left(\beta_{i-\frac{1}{2}} - \beta_{i+\frac{1}{2}}\right) + \frac{1}{2}(x_1 - x_2)\left(\beta_{i-\frac{1}{2}} - \beta_{i+\frac{1}{2}}\right)$$

$x_1$ and $x_2$ denote the ratios of the interfaces on the top and bottom of the cell.. the desired value is the average of these ratios: $\frac{1}{2}(x_1 + x_2)$.

Solving:

$$\frac{\alpha_{i,j} - \beta_{i+\frac{1}{2},j}}{\beta_{i-\frac{1}{2}} - \beta_{i+\frac{1}{2}}} = x_2 + \frac{1}{2}(x_1 - x_2) = \frac{1}{2}(x_1 + x_2)$$

The last term needed for the total two-dimensional cut-cell flux is found by applying the one-dimensional flux to the shielded portion of the cell to give

$$\mathbf{f}^{shielded}_{i-\frac{1}{2},j} = \mathbf{f}^{*}_{i-\frac{1}{2},j}\left(\alpha^{shielded}_{i-\frac{1}{2},j}\right)$$

The generic idea of fluxes would need to be applied to the CFD equations.

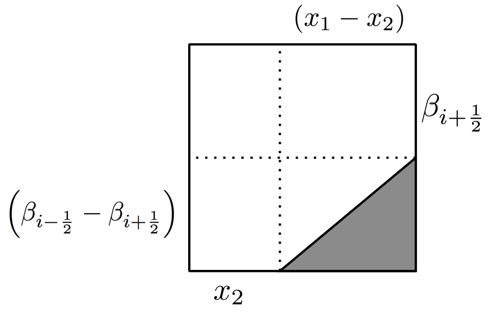
$(x_1 - x_2)$
$\beta_{i+\frac{1}{2}}$
$\left(\beta_{i-\frac{1}{2}} - \beta_{i+\frac{1}{2}}\right)$
$x_2$

Figure 12: Labeled cut cell for derivation of ratio expression for finding average distance to boundary

### 4.3.3 Extension to Pressure/other considerations

This paper accounts for the changes in calculations of normal boundary fluxes, i.e. the advective flux, which has been altered accordingly. This is actually a lot easier than the advective and diffusive fluxes. Following the derivation presented in Harlow[3] for an incompressible fluid on a regular grid, the necessary modifications on the pressure terms are clearer.

As described in Section 4.2, the pressure terms are solved for implicitly. For an incompressible fluid, the density $\rho$ should remain constant and thus there is no overall change in mass in the cell: $\Delta mass_{total} = 0$. The change in mass over each boundary is described by $\Delta mass = (Flux)\,(Area)\,(dt)$. The cut cell affects the area, so the sum can be adjusted as

$$\begin{aligned}\Delta mass_{total} =&\\ &\left(\rho u_{i-\frac{1}{2},j}\right)\left(\beta_{i-\frac{1}{2},j} dy \cdot dz\right)(dt)\\ &-\left(\rho u_{i+\frac{1}{2},j}\right)\left(\beta_{i+\frac{1}{2},j} dy \cdot dz\right)(dt)\\ &+\left(\rho v_{i,j-\frac{1}{2}}\right)\left(\beta_{i,j-\frac{1}{2}} dx \cdot dz\right)(dt)\\ &-\left(\rho v_{i,j+\frac{1}{2}}\right)\left(\beta_{i,j+\frac{1}{2}} dx \cdot dz\right)(dt)\\ &\qquad\qquad\qquad = 0\end{aligned}$$

Factoring out the constant terms and dividing through by $dx \cdot dy$:

$$\rho \cdot dz \cdot dt\left[\frac{u_{i-\frac{1}{2},j}\beta_{i-\frac{1}{2},j} - u_{i+\frac{1}{2},j}\beta_{i+\frac{1}{2},j}}{dx} + \frac{v_{i,j-\frac{1}{2}}\beta_{i,j-\frac{1}{2}} - v_{i,j+\frac{1}{2}}\beta_{i,j+\frac{1}{2}}}{dy}\right] = 0$$

As the constants cannot be 0, the expression $D_{i,j}$ to be minimized to 0 is

$$D_{i,j} = \frac{u_{i-\frac{1}{2},j}\beta_{i-\frac{1}{2},j} - u_{i+\frac{1}{2},j}\beta_{i+\frac{1}{2},j}}{dx} + \frac{v_{i,j-\frac{1}{2}}\beta_{i,j-\frac{1}{2}} - v_{i,j+\frac{1}{2}}\beta_{i,j+\frac{1}{2}}}{dy}$$

Furthermore, to use Newton's method in solving for pressure, the value for the next iteration is given as

$$P^{new} = P^{old} - \frac{D\left(P^{old}\right)}{\left(\frac{\delta D}{\delta P}\right)_{i,j}}$$

The value for the partial derivative can be found through the general expression for the velocity update

$$u^{n+1}_{i+\frac{1}{2},j} = \bar{u}^{n+1}_{i+\frac{1}{2},j} - dt\left(\frac{P^{n+1}_{i+1,j} - P^{n+1}_{i,j}}{dx}\right)$$

$$v^{n+1}_{i,j+\frac{1}{2}} = \bar{u}^{n+1}_{i,j+\frac{1}{2}} - dt\left(\frac{P^{n+1}_{i,j+1} - P^{n+1}_{i,j}}{dy}\right)$$

and application of the chain rule. Applying the chain rule yields

$$\begin{aligned}\left(\frac{\delta D}{\delta P}\right)_{i,j} &= \left(\frac{\delta D_{i,j}}{\delta u_{i+\frac{1}{2},j}}\right)\left(\frac{\delta u_{i+\frac{1}{2},j}}{\delta P_{i,j}}\right) + \left(\frac{\delta D_{i,j}}{\delta u_{i-\frac{1}{2},j}}\right)\left(\frac{\delta u_{i-\frac{1}{2},j}}{\delta P_{i,j}}\right) + \left(\frac{\delta D_{i,j}}{\delta v_{i,j+\frac{1}{2}}}\right)\left(\frac{\delta v_{i,j+\frac{1}{2}}}{\delta P_{i,j}}\right) + \left(\frac{\delta D_{i,j}}{\delta v_{i,j-\frac{1}{2}}}\right)\left(\frac{\delta v_{i,j-\frac{1}{2}}}{\delta P_{i,j}}\right)\\ &= \left(\frac{\beta_{i-\frac{1}{2},j}}{dx}\right)\left(\frac{dt}{dx}\right) + \left(-\frac{\beta_{i+\frac{1}{2},j}}{dx}\right)\left(-\frac{dt}{dx}\right) + \left(\frac{\beta_{i,j-\frac{1}{2}}}{dy}\right)\left(\frac{dt}{dy}\right) + \left(-\frac{\beta_{i,j+\frac{1}{2}}}{dy}\right)\left(-\frac{dt}{dy}\right)\\ &= dt\left[\frac{\beta_{i-\frac{1}{2},j} + \beta_{i+\frac{1}{2},j}}{dx^2} + \frac{\beta_{i,j-\frac{1}{2}} + \beta_{i,j+\frac{1}{2}}}{dy^2}\right]\end{aligned}$$

This has not yet been implemented, but should adjust the influence of pressure on the change in velocity to account for the embedded object and weight it properly as such.

## 5 Conclusion

This project has not reached a point at which it can readily be used to test car modifications nor be validated through experimental data on this front. We have, however, outlined a feasible overall means to solving the problem and have made progress in implementing some of the necessary steps.

## Acknowledgements

We would like to thank our mentors, for helping us take the initial seed of an idea and turn it into a workable project, as well as for supplying help, or directions for where to find help. We would also like to thank Mr. Goodwin, for helping keep us organized.

## References

[1] AN-31 Op Amp Circuit Collection. Tech. no. SNLA140A. Texas Instruments, May 2004. Web. 2 Apr. 2013.

[2] "Argonne GREET Model." Argonne GREET Model. Office of Energy Efficiency and Renewable Energy, 3 Sept. 2010. Web. 02 Apr. 2013.

[3] E. Scannapieco and F. H. Harlow. Introduction to Finite-Difference Methods for Numerical Fluid Dynamics. LA-12984(UC-700), 1995.

[4] Ivan Sutherland, Gary W. Hodgman: Reentrant Polygon Clipping. Communications of the ACM, vol. 17, pp. 32-41, 1974.

[5] Iwilltry. "Measure the Drag Coefficient of Your Car." Instructables.com. Instructables, 30 Aug. 2007. Web. 09 Feb. 2013.

[6] R. Klein, K. R. Bates, N. Nikiforakis. (2009) Well-balanced compressible cut-cell simulation of atmospheric flow. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 367:1907, pp. 4559-4575.

[7] SketchUp Class Index. <http://www.sketchup.com/intl/en/developer/docs/classes>

[8] Squatriglia, Chuck. "Go Green — Buy a Used Car. It's Better Than a Hybrid." Wired.com. Wired, 19 May 2008. Web. 02 Apr. 2013.

[9] Turner, Mike. "Modifying a Honda Civic for Maximum MPG." Aerocivic. N.p., Oct. 2011. Web. 09 Feb. 2013.

[10] UC Irving. Computational Geometry Lecture Notes. 7 March 1996. <http://www.ics.uci.edu/~eppstein/161/960307.html>

[11] University of Cambridge. Cut-cell/Embedded boundary techniques. Laboratory for Scientific Computing, 2012 <http://www.lsc.phy.cam.ac.uk/ research/cutcells.shtml>

# A Code Listings

## A.1 Ruby Script CSV Export

```
require 'sketchup.rb'
def exportCSV()
     sep=","
     ext="csv"
     model = Sketchup.active_model
     ss = model.selection
     faces = []
     ss.each{|e|faces << e if e.class == Sketchup::Face}
     if not faces
          UI.messagebox("No faces selected.\nExiting.\n")
          return nil
     end

     path=model.path
     if not path or path==""
          path=Dir.pwd
          title="Untitled"
     else
          path=File.dirname(path)
          title=model.title
     end#if
     ofile=File.join(path,title+'_verts.'+ext).tr("\\","/")
     begin
          file=File.new(ofile,"w")
          rescue### trap if open
          UI.messagebox("Vertices File:\n\n  "+ofile+"\n\nCannot be written - it's probably ↩
              already open.\nClose it and try making it again...\n\nExiting...")
          return nil
     end

     pts=[]
     faces.each{|face|
          verts = face.vertices
          pts.clear
          verts.each{|v|pts << v.position.x.to_s.gsub(/^~ /,'').to_f.to_s+sep+v.position.y.to_s.↩
              gsub(/^~/,'').to_f.to_s+sep+v.position.z.to_s.gsub(/^~/,'').to_f.to_s}
          file.puts(pts.length.to_s+"\n")
          pts.each{|pt|file.puts(pt)}
     }
     file.close
     #puts (faces.length.to_s)+" vertices written to\n"+ofile
     begin
          UI.openURL("file:/"+ofile)
          rescue
     end
end
###
```

## A.2 Mesh Generation

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <OpenGL/gl.h>
#include <GLUT/glut.h>

#define idx(i,j) ((i)*ysize+(j))
#define MIN(x,y) (x < y ? x : y)
#define MAX(x,y) (x > y ? x : y)
#define AVG(a,b) (((a)+(b))/2.)
#define EPS 1.0e-6

#define INSIDE 1
#define OUTSIDE 0

int window, height = 600, width = 600;

typedef struct {
     float x, y;
```

```
}POINT;
typedef struct {
     int nverts;
     POINT *vertices;
}POLYGON;


float left, right, bottom, top;    //bounds of overlay grid
float dx = 0.5, dy = 0.5, dz = 1.0;
int ncells, xsize, ysize;
POLYGON *grid;
POLYGON poly;

/************************************************
 * Initialization routines
 ************************************************/
void importPolygon(char* file_name) {
     FILE* file = fopen(file_name, "r");
     if( file == NULL ) {
          printf("Error opening file\n");
          exit(0);
     }
     char str[80];
     int i;

     /* Parse 2D coordinates */
     fgets(str, 80, file);
     sscanf(str, "%d", &poly.nverts);
     poly.vertices = malloc(poly.nverts*sizeof(POINT));

     i = 0;
     while( fgets(str, 80, file) != NULL && i < poly.nverts ) {
          sscanf(str, "%f, %f", &poly.vertices[i].x, &poly.vertices[i].y);
          i++;
     }
     fclose(file);

}
void genGrid() {
     int v;
     float xmax = poly.vertices[0].x,
            xmin = poly.vertices[0].x,
            ymax = poly.vertices[0].y,
            ymin = poly.vertices[0].y;
     for( v = 1; v < poly.nverts; v++ ) {
          if( poly.vertices[v].x < xmin ) xmin = poly.vertices[v].x;
          if( poly.vertices[v].x > xmax ) xmax = poly.vertices[v].x;
          if( poly.vertices[v].y < ymin ) ymin = poly.vertices[v].y;
          if( poly.vertices[v].y > ymax ) ymax = poly.vertices[v].y;
     }
     left = xmin-dx;
     right = ((int)((xmax-xmin)/dx+0.5) + 2)*dx + xmin;
     bottom = ymin - dy;
     top = ((int)((ymax-ymin)/dy+0.5) + 2)*dy + ymin;

     xsize = (right-left)/dx;
     ysize = (top-bottom)/dy;
     ncells =  xsize * ysize;

     grid = malloc(ncells * sizeof(POLYGON));
     int i, j;

     for( i = 0; i < xsize; i++ ) {
          for( j = 0; j < ysize; j++ ) {
               grid[idx(i,j)].nverts = 4;
               grid[idx(i,j)].vertices = malloc(4*sizeof(POINT));
               grid[idx(i,j)].vertices[0].x = i*dx + left;
               grid[idx(i,j)].vertices[0].y = j*dy + bottom;
               grid[idx(i,j)].vertices[1].x = (i+1)*dx + left;
               grid[idx(i,j)].vertices[1].y = j*dy + bottom;
               grid[idx(i,j)].vertices[2].x = (i+1)*dx + left;
               grid[idx(i,j)].vertices[2].y = (j+1)*dy + bottom;
               grid[idx(i,j)].vertices[3].x = i*dx + left;
               grid[idx(i,j)].vertices[3].y = (j+1)*dy + bottom;
          }
     }
}
/************************************************
 * Clipping utility routines
 ************************************************/
int sameSign(float x, float y) {
     return ((x*y) >= 0.);
}
float getLineXcoor(POINT a, POINT b, float ycoor) {
     if( a.x == b.x ) return a.x;  //check vertical line

     double slope = (a.y - b.y)/(a.x - b.x);
     double y_incpt = b.y - (b.x * slope);
     return (float)(ycoor-y_incpt)/slope;
}
float getLineYcoor(POINT a, POINT b, float xcoor) {
     if( a.y == b.y ) return a.y;

     double slope = (a.y - b.y)/(a.x - b.x);
     double y_incpt = b.y - (b.x * slope);
     return (float)(slope*xcoor+y_incpt);
}
/* Test to see if the point is in the polygon, set on_boundary to desired return value for ↩
    points lying on polygon edge */
int pnpoly(POINT *polygon, int n, POINT P, int on_boundary) {
     int v, crossings = 0;
     POINT ic, next, prev;
     float t, cy;

     for( v = 0; v < n; v++ ) {
          ic = polygon[v];
          if( v == n-1 ) next = polygon[0];
          else next = polygon[v+1];
          if( v == 0 ) prev = polygon[n-1];
          else prev = polygon[v-1];

          if( (ic.x < P.x && P.x < next.x) || (ic.x > P.x && P.x > next.x) ) {
               t = (P.x - next.x) / (ic.x - next.x);
               cy = t*ic.y + (1. - t)*next.y;
               if( fabs(P.y - cy) < EPS ) {
                    return on_boundary;
               }
               else if( P.y > cy ) crossings++;
          }
          if( ic.x == P.x && ic.y <= P.y ) {
               if( ic.y == P.y) return on_boundary;
               if( next.x == P.x ) {
                    if( (ic.y <= P.y && P.y <= next.y) || (ic.y >= P.y && P.y >= next.y) ) {
                         return on_boundary;
                    }
               }
               if( prev.x == P.x ) {
                    if( (ic.y <= P.y && P.y <= prev.y) || (ic.y >= P.y && P.y >= prev.y) ) {
                         return on_boundary;
                    }
               }
               else if( !sameSign(ic.x - next.x, ic.x - prev.x) ) crossings++;
          }
     }
     if( crossings % 2 == 0 ) return OUTSIDE;
     else return INSIDE;

}
/* Check if point P lies between the x and y values of a and b */
int pnrange(POINT a, POINT b, POINT P) {

     if( (a.x <= P.x && P.x <= b.x) || (a.x >= P.x && P.x >= b.x) ) {
          if( (a.y <= P.y && P.y <= b.y) || (a.y >= P.y && P.y >= b.y) ) {
               return 1;
          }
     }
     return 0;
}
int lineIntersect(POINT *polygon, int n, POINT a, POINT b) {
     int v;
     double x, y;
     POINT pa, pb;
     double m1, m2, b1, b2;   //variables for 2 lines

     for( v = 0; v < n; v++ ) {
          pa = polygon[v];
          if( v == n-1 ) pb = polygon[0];
          else pb = polygon[v+1];

          if( pa.x == pb.x ) {
               y = getLineYcoor(a, b, pa.x);
               if( (y <= pa.y && y >= pb.y) || (y >= pa.y && y <= pb.y) )
```

```
                    return 1;
          }
          else if( pa.y == pa.y ) {
               x = getLineXcoor(a, b, pa.y);
               if( (x <= pa.x && x >= pb.x) || (x >= pa.x && x <= pb.x) )
                    return 1;
          }
          else {

               m1 = (pa.y - pb.y)/(pa.x - pb.x);
               m2 = (a.y - b.y)/(a.x - b.x);
               b1 = pa.y - (pa.x * m1);
               b2 = a.y - (a.x * m2);


               y = (m2 / (m2 - m1))*b1 - (m1/m2)*b2;

               //compare x values to decide if lines intersect
               if( fabs((y-b1)/m1 - (y-b2)/m2) < EPS ) {
                    return 1;
               }
          }
     }
     return 0;
}
void purgeInner() {
     int i, v;
     int allInside;
     for( i = 0; i < ncells; i++ ) {
          allInside = 1;
          for( v = 0; v < grid[i].nverts; v++ ) {
               if( !pnpoly(poly.vertices, poly.nverts, grid[i].vertices[v], INSIDE) )
                    allInside = 0;
          }
          if( allInside ) {
               grid[i].nverts = 0;
               free(grid[i].vertices);
          }
     }
}
void clip(POINT a, POINT b, POINT c) {
     int i, j, v;
     POINT P, S, I;
     double dist1, dist2, part_line;
     POINT *clipverts;
     int ncverts;

     for( i = (int)((MIN(a.x,b.x)-left)/dx); i <= (int)((MAX(a.x,b.x)-left)/dx); i++ ) {
          for( j = (int)((MIN(a.y,b.y)-bottom)/dy); j <= (int)((MAX(a.y,b.y)-bottom)/dy); j++ ) ↩
              {

               clipverts = malloc(20*sizeof(POINT));
               ncverts = 0;

               if( lineIntersect(grid[idx(i,j)].vertices, grid[idx(i,j)].nverts, a, b) ) {

                    int temp = idx(i,j);

                    for( v = 0; v < grid[idx(i,j)].nverts; v++ ) {

                         P = grid[idx(i,j)].vertices[v];
                         if( v == 0 ) S = P;
                         else {
                              if( a.y == b.y ) {
                                   dist1 = P.y - a.y;
                                   dist2 = S.y - a.y;
                              }
                              else {
                                   dist1 = P.x - getLineXcoor(a, b, P.y);
                                   dist2 = S.x - getLineXcoor(a, b, S.y);
                              }
                              //check to see if the endpoints are on different sides of clipping↩
                                   plane
                              if( !sameSign(dist1, dist2) ) {
                                   part_line = fabs(dist2)/(fabs(dist1)+fabs(dist2));
                                   I.x = part_line*(P.x - S.x) + S.x;
                                   I.y = part_line*(P.y - S.y) + S.y;
                                   if( pnrange(a, b, I) ) {
                                        clipverts[ncverts] = I;
                                        ncverts++;
                                   }
```

```
                        }

                        if( b.y == c.y ) {
                             dist1 = P.y - b.y;
                             dist2 = S.y - b.y;
                        }
                        else {
                             dist1 = P.x - getLineXcoor(b, c, P.y);
                             dist2 = S.x - getLineXcoor(b, c, S.y);
                        }
                        //check to see if the endpoints are on different sides of clipping↩
                             plane
                        if( !sameSign(dist1, dist2) ) {
                             part_line = fabs(dist2)/(fabs(dist1)+fabs(dist2));
                             I.x = part_line*(P.x - S.x) + S.x;
                             I.y = part_line*(P.y - S.y) + S.y;
                             if( pnrange(b, c, I) ) {
                                  clipverts[ncverts] = I;
                                  ncverts++;
                             }
                        }
                        S = P;          //update S for consideration of point
                   }
                   if( !pnpoly(poly.vertices, poly.nverts, S, OUTSIDE) ) {
                        clipverts[ncverts] = S;
                        ncverts++;
                   }
              }
              if( grid[idx(i,j)].nverts > 0 ) {
                   S = grid[idx(i,j)].vertices[grid[idx(i,j)].nverts-1];
                   P = grid[idx(i,j)].vertices[0];

                   if( a.y == b.y ) {
                        dist1 = P.y - a.y;
                        dist2 = S.y - a.y;
                   }
                   else {
                        dist1 = P.x - getLineXcoor(a, b, P.y);
                        dist2 = S.x - getLineXcoor(a, b, S.y);
                   }

                   if( !sameSign(dist1, dist2) ) {
                        part_line = fabs(dist2)/(fabs(dist1)+fabs(dist2));
                        I.x = part_line*(P.x - S.x) + S.x;
                        I.y = part_line*(P.y - S.y) + S.y;
                        if( pnrange(a, b, I) ) {
                             clipverts[ncverts] = I;
                             ncverts++;
                        }
                   }
                   if( b.y == c.y ) {
                        dist1 = P.y - b.y;
                        dist2 = S.y - b.y;
                   }
                   else {
                        dist1 = P.x - getLineXcoor(b, c, P.y);
                        dist2 = S.x - getLineXcoor(b, c, S.y);
                   }

                   if( !sameSign(dist1, dist2) ) {
                        part_line = fabs(dist2)/(fabs(dist1)+fabs(dist2));
                        I.x = part_line*(P.x - S.x) + S.x;
                        I.y = part_line*(P.y - S.y) + S.y;
                        if( pnrange(b, c, I) ) {
                             clipverts[ncverts] = I;
                             ncverts++;
                        }
                   }
                   grid[idx(i,j)].nverts = ncverts;
                   free( grid[idx(i,j)].vertices );
                   grid[idx(i,j)].vertices = clipverts;

              }
         }
    }
  }
}
void clipPolygon() {
     POINT next, curr, nxtnext;
     int v;
     for( v = 0; v < poly.nverts; v++ ) {
          curr = poly.vertices[v];
          if( v == poly.nverts - 1 ) next = poly.vertices[0];
          else next = poly.vertices[v+1];
          if( v == poly.nverts - 2 ) nxtnext = poly.vertices[0];
          else if( v == poly.nverts - 1) nxtnext = poly.vertices[1];
          else nxtnext = poly.vertices[v+2];
          clip(curr, next, nxtnext);
     }
}
/**************************************************
 * Geometry routines
 **************************************************/
/* Return relative volume of cell to unclipped one -- vertices must be in order */
float relVolume(POLYGON p) {

     double total = 0;
     int i;

     if( p.nverts < 1 ) return 0.;

     /* Gauss's area formula or shoelace method */
     for( i = 0; i < p.nverts-1; i++ ) {
          total += p.vertices[i].x * p.vertices[i+1].y;
          total -= p.vertices[i].y * p.vertices[i+1].x;
     }
     total += p.vertices[p.nverts-1].x * p.vertices[0].y;
     total -= p.vertices[p.nverts-1].y * p.vertices[0].x;

     total = fabs(total)/2.;
     total = ((int)(total/(dx*dy)*1000+0.5))/1000.;

     return total;

}
/* returns relative cross section area on cell interface */
float relArea(POINT a, POINT b, float dist) {
     double length = sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));
     return length/dist;
}
/**************************************************
 * Grid file output routines
 **************************************************/
void writeFile(char *file_name) {

     float vertInt[ysize][xsize+1];
     float horzInt[ysize+1][xsize];
     float relVol[ysize][xsize];


     int c,v,i,j;
     float tx,ty;

     /* Determine relative volumes */
     for( j = 0; j < ysize; j++ ) {
          for( i = 0; i < xsize; i++ ) {
               relVol[j][i] = -1.;
          }
     }

     for( c = 0; c < ncells; c++ ) {
          if( grid[c].nverts == 0 ) continue;
          tx = 0.;
          ty = 0.;
          for( v = 0; v < grid[c].nverts; v++ ) {
               tx += grid[c].vertices[v].x-left;
               ty += grid[c].vertices[v].y-bottom;
          }
          tx /= grid[c].nverts;
          ty /= grid[c].nverts;

          i = (int)(tx/dx);
          j = (int)(ty/dy);

          relVol[j][i] = relVolume(grid[c]);

     }

     /* Determine relative interface area */
     for( j = 0; j < ysize; j++ ) {
          for( i = 0; i <= xsize; i++ ) {
                vertInt[j][i] = -1.;
            }
        }

        for( j = 0; j <= ysize; j++ ) {
            for( i = 0; i < xsize; i++ ) {
                horzInt[j][i] = -1.;
            }
        }

        POINT curr, next;

        for( c = 0; c < ncells; c++ ) {
            if( grid[c].nverts == 0 ) continue;
            for( v = 0; v < grid[c].nverts; v++ ) {
                curr = grid[c].vertices[v];
                if( v == grid[c].nverts-1 ) next = grid[c].vertices[0];
                else next = grid[c].vertices[v+1];

                if( curr.x == next.x ) {
                    i = (int)((curr.x-left)/dx);
                    j = (int)((AVG(curr.y,next.y)-bottom)/dy);
                    vertInt[j][i] = relArea(curr, next, dy);
                }
                if( curr.y == next.y ) {
                    j = (int)((curr.y-bottom)/dy);
                    i = (int)((AVG(curr.x,next.x)-left)/dx);
                    horzInt[j][i] = relArea(curr, next, dy);
                }
            }

        }

        FILE *file = fopen(file_name, "w");
        fprintf(file, "%d,%d\n", xsize, ysize);
        fprintf(file, "%f,%f\n", dx, dy);

        for( j = 0; j < ysize; j++ ) {
            for( i = 0; i < xsize; i++ ) {
                fprintf(file, "%f\n", relVol[j][i]);
            }
        }
        for( j = 0; j <= ysize; j++ ) {
            for( i = 0; i < xsize; i++ ) {
                fprintf(file, "%f\n", horzInt[j][i]);
            }
        }
        for( j = 0; j < ysize; j++ ) {
            for( i = 0; i <= xsize; i++ ) {
                fprintf(file, "%f\n", vertInt[j][i]);
            }
        }

        fclose(file);
}
/************************************************
 * Display routines
 ************************************************/
void displayInit() {
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glDisable(GL_DEPTH_TEST);
        glLineWidth(2.);
}
void drawGrid() {
        glColor3f(0.0f, 0.0f, 0.0f);
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        int i, v;
        for( i = 0; i < ncells; i++ ) {
            glBegin(GL_POLYGON);
            for( v = 0; v < grid[i].nverts; v++ ) {
                glVertex2f(grid[i].vertices[v].x, grid[i].vertices[v].y);
            }
            glEnd();
        }

}
void drawPoly() {

        glColor3f(1.0f, 0.0f, 0.0f);
```

```
     int v;
     glBegin(GL_LINE_LOOP);
     for( v = 0; v < poly.nverts; v++ ) {
          glVertex2f(poly.vertices[v].x, poly.vertices[v].y);
     }
     glEnd();
}
void display() {

     displayInit();

     glClear(GL_COLOR_BUFFER_BIT);

     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     gluOrtho2D(left-dx, right+dx, bottom-dx, top+dx);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();

     drawGrid();
     drawPoly();

     glutSwapBuffers();
}
/************************************************
 * Main
 ************************************************/
int main(int argc, char *argv[]) {

     importPolygon("poly2.txt");
     genGrid();

     clipPolygon();
     purgeInner();

     writeFile("/Users/rachel/Documents/CFD/CFD/grid2.txt");

     /* Open display */
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_DOUBLE);
     glutInitWindowSize(width, height);
     window = glutCreateWindow("Polygon clipping");
     displayInit();
     glutDisplayFunc(display);

     glutMainLoop();

     return 0;
}
```

## A.3 Regular Grid CFD

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PRINT 0

#define ANU 0.06
#define DTEST 0.005
#define POS 0
#define NEG 1

#define AVG(a,b) (((a)+(b))/2.)

#define p(i,j)     p[(j)*(xsize+2)+(i)]
#define D(i,j)     D[(j)*(xsize+2)+(i)]

#define u(i,j)     u[(j)*(xsize+1)+(i)]
#define v(i,j)     v[(j)*(xsize+2)+(i)]
#define u_bar(i,j)    u_bar[(j)*(xsize+1)+(i)]
#define v_bar(i,j)    v_bar[(j)*(xsize+2)+(i)]

#define psi(i,j)      psi[(j)*(xsize+2)+(i)]

double *u_bar, *v_bar;
double *p, *D;
double *u, *v;

double *psi;

FILE *fpsi;
FILE *uFile, *vFile;

int xsize, ysize;

int setObj() {
     int j;
     for( j = (int)(ysize/3)+1; j <= ysize - (int)(ysize/3); j++ ) {
          u(6,j) = 0.0;
     }
     return 0;
}

int runCFD() {

     xsize = 18; ysize = 12;
     double dx = 1., dy = 1.; //cm

     double dt = 0.001;    //s
     double ttime = 5.003, time = 0., ptime = 5., pt = 0.;

     int i, j;
     int iter;
     double beta = 1/(2*dt*(1/dx/dx+1/dy/dy));
     double dmax;
     double insum, outsum;

     p       = malloc((xsize+2)*(ysize+2)*sizeof(double));
     D       = malloc((xsize+2)*(ysize+2)*sizeof(double));

     u       = malloc((xsize+1)*(ysize+2)*sizeof(double));
     v       = malloc((xsize+2)*(ysize+1)*sizeof(double));
     u_bar   = malloc((xsize+1)*(ysize+2)*sizeof(double));
     v_bar   = malloc((xsize+2)*(ysize+1)*sizeof(double));

     psi   = malloc((xsize+1)*(ysize+1)*sizeof(double));

     /* Initialization */

     for( i = 0; i <= xsize; i++ ) {
          for( j = 1; j <= ysize; j++ ) {
               u(i,j) = 0.0;
          }
     }
     for( i = 1; i <= xsize; i++ ) {
          for( j = 0; j <= ysize; j++ ) {
               v(i,j) = 0.0;
          }
     }
     for( i = 0; i <= xsize+1; i++ ) {
          for( j = 0; j <= ysize+1; j++ ) {
               p(i,j) = 0.0;
          }
     }

/************************
 * Iteration Loop
 ************************/
     for( time = 0.; time <= ttime; time += dt ) {
          if(PRINT) printf("time %.3lf\n", time);

          /* Boundary Conditions */
          //sum input
          insum = 0.; outsum = 0.;
          for( j = 1; j <= ysize; j++ ) {
               //free slip
               v(0,j-1) = v(1,j-1);
               v(xsize+1,j-1) = v(xsize,j-1);

               u(0,j) = 1.0;

               insum += u(0,j);
               outsum += u(xsize-1,j);
               if( outsum == 0. ) outsum = (float)ysize;
          }
```

```
for( j = 1; j <= ysize; j++ ) {
     u(xsize,j) = u(xsize-1,j)*(insum/outsum);
}
for( i = 1; i <= xsize; i++ ) {
     //free slip
     u(i-1,0) = u(i-1,1);
     u(i-1,ysize+1) = u(i-1,ysize);

     //no output on vertical
     v(i,0) = 0.0;
     v(i,ysize) = 0.0;
}
if(PRINT) printf("\tBoundary Conditions Complete\n");
setObj();
if(PRINT) printf("\tObject set\n");

/* Velocities -- cell centered */
for( j = 1; j <= ysize; j++ ) {
     for( i = 0; i <= xsize; i++ ) {
          u_bar(i,j) = u(i,j) - dt*(
                 (pow(AVG(u(i,j),u(i+1,j)),2)-pow(AVG(u(i-1,j),u(i,j)),2))/dx
               + (AVG(u(i,j),u(i,j+1))*AVG(v(i,j),v(i+1,j))-AVG(u(i,j-1),u(i,j))*AVG(v↩
                   (i,j-1),v(i+1,j-1))))/dy
               - ANU*(u(i+1,j)+u(i-1,j)-2*u(i,j))/(dx*dx)
               - ANU*(u(i,j+1)+u(i,j-1)-2*u(i,j))/(dy*dy) );
     }
}
for( j = 0; j <= ysize; j++ ) {
     for( i = 1; i <= xsize; i++ ) {
          v_bar(i,j) = v(i,j) - dt*(
                 (pow(AVG(v(i,j),v(i,j+1)),2)-pow(AVG(v(i,j-1),v(i,j)),2))/dy
               + (AVG(u(i,j),u(i,j+1))*AVG(v(i,j),v(i+1,j))-AVG(u(i-1,j),u(i-1,j+1))*↩
                   AVG(v(i-1,j),v(i,j)))/dx
               - ANU*(v(i+1,j)+v(i-1,j)-2*v(i,j))/(dx*dx)
               - ANU*(v(i,j+1)+v(i,j-1)-2*v(i,j))/(dy*dy) );
     }
}

if(PRINT) printf("\tVelocity init calculated\n");

/* Pressure */
iter = 0;
dmax = 0.0;
do {
     for( j = 1; j <= ysize; j++ ) {
          for( i = 1; i <= xsize; i++ ) {
               D(i,j) = (u(i,j)-u(i-1,j))/dx + (v(i,j)-v(i,j-1))/dy;
               if( fabs(D(i,j)) > dmax ) dmax = fabs(D(i,j));
          }
     }
     for( j = 1; j <= ysize; j++ ) {
          for( i = 1; i <= xsize; i++ ) {
               p(i,j) = p(i,j) - (beta*D(i,j));
          }
     }
     for( j = 1; j <= ysize; j++ ) {
          for( i = 0; i < xsize; i++ ) {
               u(i,j) = u_bar(i,j) + (dt/dx)*(p(i,j)-p(i+1,j));
          }
     }
     for( j = 0; j < ysize; j++ ) {
          for( i = 1; i <= xsize; i++ ) {
               v(i,j) = v_bar(i,j)+(dt/dy)*(p(i,j)-p(i,j+1));
          }
     }
     /* Boundary Conditions */
     //sum input
     insum = 0.; outsum = 0.;
     for( j = 1; j <= ysize; j++ ) {
          //free slip
          v(0,j-1) = v(1,j-1);
          v(xsize+1,j-1) = v(xsize,j-1);

          u(0,j) = 1.0;

          insum += u(0,j);
          outsum += u(xsize-1,j);
          if( outsum == 0. ) outsum = (float)ysize;
     }

     for( j = 1; j <= ysize; j++ ) {
```

```
                u(xsize,j) = u(xsize-1,j)*(insum/outsum);
            }
            for( i = 1; i <= xsize; i++ ) {
                //free slip
                u(i-1,0) = u(i-1,1);
                u(i-1,ysize+1) = u(i-1,ysize);

                //no output on vertical
                v(i,0) = 0.0;
                v(i,ysize) = 0.0;
            }

            setObj();

            iter++;


        } while((dmax > DTEST) && (iter < 100));

        if(PRINT) printf("\tPressure complete\n");

        /* Output */
        if( pt >= ptime ) {

            if(PRINT) printf("Time %.2lf\n", time);

            uFile = fopen("/Users/rachel/SCC13/Python/u.txt", "w");
            for( j = 1; j <= ysize; j++ ) {
                for( i = 0; i < xsize; i++ ) {

                    fprintf(uFile, "%lf", (u(i,j)+u(i+1,j))/2.);
                    if(i!=xsize-1) fprintf(uFile,",");

                }
                fprintf(uFile, "\n");
            }
            fclose(uFile);

            vFile = fopen("/Users/rachel/SCC13/Python/v.txt", "w");
            for( j = 0; j < ysize; j++ ) {
                for( i = 1; i <= xsize; i++ ) {

                    fprintf(vFile, "%lf", (v(i,j)+v(i,j+1))/2.);
                    if(i!=xsize) fprintf(vFile,",");

                }
                fprintf(vFile, "\n");
            }
            fclose(vFile);

            fpsi = fopen("/Users/rachel/SCC13/Python/psi.txt", "w");

            for( i = 0; i <= xsize; i++ ) {
                psi(i,0) = 0.;
            }

            for( i = 0; i <= xsize; i++ ) {
                for( j = 1; j <= ysize; j++ ) {
                    psi(i,j) = psi(i,j-1) + dy*(u(i,j));
                }
            }

            for( j = 0; j <= ysize; j++ ) {
                for( i = 0; i <= xsize; i++ ) {

                    fprintf(fpsi, "%lf", psi(i,j));
                    if(i!=xsize) fprintf(fpsi,",");

                }
                fprintf(fpsi, "\n");
            }
            fclose(fpsi);

            pt = 0.;
        }
        pt+= dt;
        if(PRINT) printf("\tOutput loop complete\n");

    }

    return 0;
```

```
}

int main(int argc, char *argv[]) {

    runCFD();

    return 0;

}
```

## A.4 Python/Matplotlib Output File Visualization

### Contour/Streamlines

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


#Read File
file = open('psi.txt')
str = file.readlines()
file.close()
psi = [line.strip('\n').split(",") for line in str]
for y in psi:
   for x in y:
      x = float(x)

#Make plot
levs = np.linspace(0,12,24)
plt.figure()
plt.title('Streamlines')
CS = plt.contour(psi,levs,colors='k')
plt.show()
```

### Vector Field

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

#Read File
file = open('u.txt')
str = file.readlines()
file.close()

U = [line.strip('\n').split(",") for line in str]
u = np.array(U,dtype='float64')

file = open('v.txt')
str = file.readlines()
file.close()

V = [line.strip('\n').split(",") for line in str]
v = np.array(V, dtype='float64')

#Make plot
fig = plt.figure()
plt.title('Vector Field')
P = plt.quiver(u,v)
plt.ylim([-1,12])
plt.xlim([-1,18])
plt.show()
```