# PotentialFlow.py a Potential Flow Solver and Visualizer

Mechanical Engineering Technical Report 2015/08
Ingo Jahn
School of Mechanical and Mining Engineering
The University of Queensland.

July 25, 2016

**Abstract**

Potential_Flow.py is a simple teaching and analysis tool for 2-D Potential Flow. It is a collection of code, that allows the construction of simple flow fields that meet the Potential Flow governing Equations. A range of plotting and visulisation tools are included.

This report is a brief userguide and example book.

# 1 Introduction

Potential Flow is a simple but powerful analysis approach to simulate inviscid flow. This report is the userguide for Potential_Flow.py a tool to analyse simple 2-D flows together with a selection of plotting and post-processing tools. The code allows the flow-fields consisting of the following building blocks to be analysed: Uniform Flow, Sink/Source, Irrotational Vortex, Doublet.
The post-processing tool allows the generation of streamline plots, velocity contour plots, and pressure contours. In addition post-processing tools are included to extract point data and data along user-defined lines.

## 1.1 Compatibility

Potential_Flow.py is written in python. The following packages are required:

- python 2.7 - any standard distribution
- numpy
- matplotlib

## 1.2 Citing this tool

When using the tool in simulations that lead to published works, it is requested that the following works are cited:

- Jahn, I. (2015), PotentialFlow.py a Potential Flow Solver and Visualizer, *Mechanical Engineering Technical Report 2015/08*, The University of Queensland, Australia

## 2 Distribution and Installation

`Potential_Flow.py` is distributed as part of the code collection maintained by the *CFCFD Group* at the University of Queensland [1]. This collection is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. This program collection is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details `http://www.gnu.org/licenses/`. Alternatively the code is included in the Appendix.

### 2.1 Modifying the code

The working version of `Potential_Flow.py` is installed in the `$HOME/e3bin` directory. If you perform modifications or improvements to the code please submit an updated version together with a short description of the changes to the authors. Once reviewed the changes will be included in future versions of the code.

# 3 Using the Tool: 5-minute version for experienced python Users

## 3.1 5-minute version for experienced python Users

If you understand python, including classes and know how the potential flow building blocks work, this is for you.

1. Find the `if __name__ == "__main__":` section of file and then adjust the following parts.
2. Create a list of instances of the various building block classes (e.g. `A1 = Uniform(1.0,)` ). A full list of options is available in section 4.1.1.
3. Create an instance of the FlowField class. `T = PlotPotentialFlow()`
4. (Optional) Adjust the size of the flow-field. `T.size(x0=0.0, x1=1.0, y0=0.0, y1 = 1.0)`
5. Solve the flow-field. `T.calc([List],n=100)`, where `[List]` is a list of building block instances from step 2.
6. Plot the results using private functions of the FlowField class (e.g. `T.plotStreamlines()`) (make sure `plt.show()` is included to display graphs)
7. Evaluate point data or extract line data
8. Run using the command: `python Potential_Flow.py`
   Filename may need to be adjusted to incorporate version)

# 4 Using the Tool: Detailed

## 4.1 Creating your Flow field

In potential flow different flow feature *building blocks*, that full-fill Laplace's equation by themselves, are superimposed (added) in order to generate complex flow-field solutions. The first part of involves setting creating such building blocks that can be combined to create a complex flow-field.

### Step: 1

Find `if __name__ == "__main__":`, the part of the file that will be executed if the file is run from the command line.

### Step: 2

Create a list of building blocks that you want to use for your flow. The result should look something like the following for Uniform Flow + a Source:

```
if __name__ == "__main__":

    # List of Building Blocks
    # Uniform Flows
    A1 = UniformFlow(5.,0.)
```

```
# Sources
D1 = Source(0.5,0., 5.)
```

The possible options for building blocks, together with detailed descriptions are described in section 4.1.1.

### 4.1.1 Building Blocks

Currently the following Building Blocks are supported.

**Uniform Flow:** `UniformFlow(u,v)`

This creates a uniform flow with the velocity components $u$ and $v$ in the x- and y-direction respectively. The streamlines for the flow-field are shown in Fig. 1a.
The streamfunction is defined as:

$$\Psi = u\,y - v\,x \tag{1}$$

**Source:** `Source(x0,y0,m)`

This generates a source (use -ve $m$ for sink) located at the position defined by $(x0, y0)$. Streamlines for the flow-field are shown Fig. 1b.
The streamfunction is defined as:

$$\theta = \tan^{-}1\left(\frac{y-y0}{x-x0}\right) \tag{2}$$

$$\Psi = \theta\frac{m}{2\,\pi} \tag{3}$$

**Vortex:** `Vortex(x0,y0,K)`

This generates an irrotational vortex of strength $K$, with the core locates at $(x0, y0)$. Streamlines for the flow-field are shown Fig. 1c.
The streamfunction is defined as:

$$r = \left[(x-x0)^2 + (y-y0)^2\right]^{\frac{1}{2}} \tag{4}$$

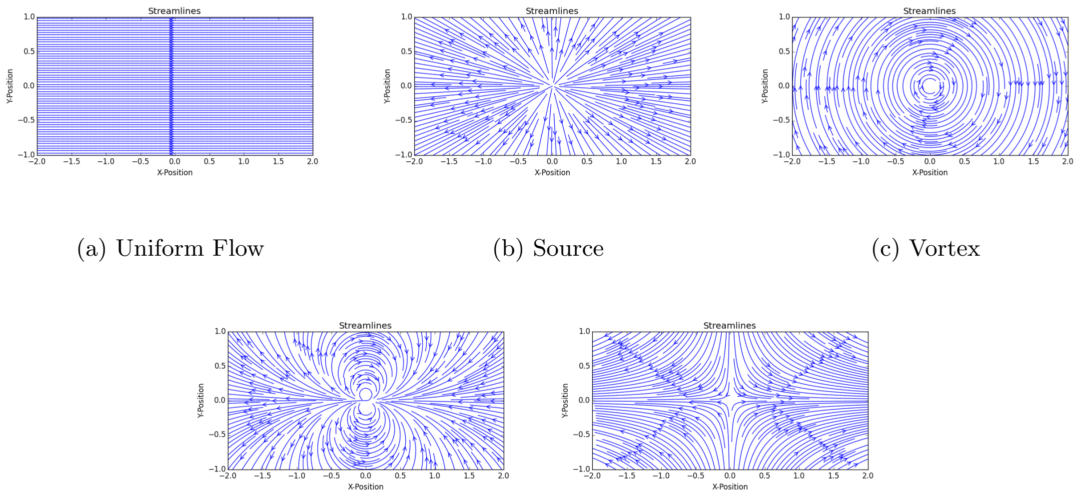
$$\Psi = -K\,\ln r \tag{5}$$

**Doublet:** `Doublet(x0,y0,a,U_inf)`

This generates the flow field known as a doublet. This is generated if a source and sink are brought very close together with a separation $s = \frac{a^2\,\pi U_\infty}{m}$, where the $\pm m$ is the strength of the source / sink. The center of the doublet is located at $(x0, y0)$. Streamlines for the flow-field are shown Fig. 1d.
The streamfunction is defined as:

$$\Psi = U_\infty\,(y-y0)\,\frac{-a^2}{(x-x0)^2 + (y-y0)^2} \tag{6}$$

This doublet works only for flow in the $+x$ directions. For other flows modify the code or manually generate a doublet by bringing together a sourcesink aligned with the flow direction.




(a) Uniform Flow (b) Source (c) Vortex

(d) Doublet (e) User Defined

Figure 1: Building Blocks available to generate Potential Flow solutions.

**User_defined:** `User_Defined(x0,y0,A)`
This generates the streamlines for flow around a 90° corner, located at position. Streamlines for the flow-field are shown Fig. 1e.
The streamfunction is defined as:

$$\Psi \quad = \quad A\ (x - x0)\ (y - y0) \tag{7}$$

**Name** `Name(x0,y0,Var1,Var2,Var3)`
This is a template for future building blocks that need to be implemented. The block class need to have the following three functions:

- `__init__(self,....)` Which is used to initialize the function
- `evaPl(self,x,y)` Which returns the value of $\Psi$ at the point defined by the coordinates $(x0, y0)$
- `eval(self,x,y)` Which returns the value of the $u$ and $v$ velocity at the point defined by the coordinates $(x0, y0)$. This should be the analytical solution to $\frac{d\Psi}{dy}$ and $-\frac{d\Psi}{dx}$.

## 4.2 Creating the Flow-Field and calculating PSI, u and v

After the building blocks have been defined, the next step is to create a flow-field area over which the Potential Flow functions will be evaluated. And to perform calculations to obtain $\Psi$, $u$, and $v$ over this field.

# Step: 3

Create an instance of the PotentialFlow-field class, set the size. The results for an area ranging from $x = -2.0$ to $x = 2.0$ and $y = -1.0$ to $x = 1.0$ should look something like:

```
# Initialise instance of Plotting Function
T = PlotPotentialFlow()     # create instance of the PotentialFlow-field class
# Set dimensions of Plotting area
T.size(-2.0, 2.0, -1.0 ,1.0)    #(x_min, x_max, y_min, y_max)
```

## Step: 4

Assemble a list of building blocks and evaluate these over an $N \times N$ grid spanning the area set in step 3. The list of blocks is generated as `[ BLK-1, BLK-2, BLK-3 ]`, where `BLK-N` are the variable names of the different blocks. For flow-field consisting of Uniform Flow + a Source (as per step 2) that is evaluated over a $100 \times 100$ grid the code is: (extent of the grid is set in step 3)

```
# Evaluate PotentialFlow-field over a grid
T.calc([A1,D1],n=100)   # ([List of elements], level of discretisation)
```

### 4.3 Plotting data

Once the flow-field has been calculated, it is possible to plot fluid properties over the flow-field area defined in step 3.

## Step: 5

Plotting commands are exercised on the Flow-field class (e.g. `T`) in the above example. To plot streamlines and a second graph of streamlines overlayed with velocity magnitude contours, use the following code. The results are shown in Fig. 2

```
# plot Data over flow-field area
T.plot_Streamlines()     # create Streamline plot.
T.plot_Streamlines_magU(100) # create plot of Streamlines + velocity magnitude

# Make sure plots are displayed on the screen
plt.show()
```

The entire program is executed using the command `python Potential_Flow.py` from the command line.

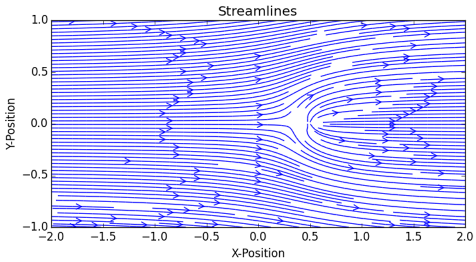
The possible options for plotting field data are described in section 4.3.1.

#### 4.3.1 Plotting Functions

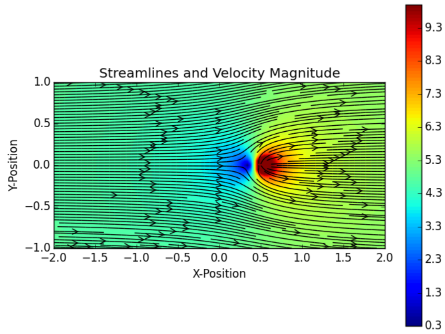
The following functions extract data from the total flow-field. They must be executed on the PotentialFlow-filed class (e.g. `T`) in the example above. In the description below `NAME` is used as a generic placeholder. These functions must be executed after `NAME.calc([List])`

**Streamfunction** `NAME.plot_Psi(levels = 40)`

Creates contours of constant streamfunction $\Psi$. `levels` sets the number of contour lines




(a) Streamlines




(b) Streamlines superimposed with contours of Velocity magnitude (magnitude capped at $10\,\mathrm{m\,s^{-1}}$).

Figure 2: Streamline and Streamline + Velocity magnitude plots generated for flow-field generated by *UniformFlow* and *Source*

shown. (Note when plotting sources/sinks a jump in $\Psi$ exists at $\theta = \pm\pi$. This can result in misleading values.

**Streamfunctions + Contours** `NAME.plot_Psi_contours( levels = 20)`
Same as `NAME.plot_Psi`, but coloured contours of constant streamfunction are added.

**Streamfunctions + Velocity** `NAME.plot_Psi_magU(magUmax = 10., levels = 20)`
Same as `NAME.plot_Psi`, but coloured contours of velocity are added. `magUmax` sets the upper limit for the velocity contours.

**Streamfunctions + Pressure** `NAME.plot_Psi_P(P_inf=0., rho=1.225, P_min=-30., P_max = 30., levels`
Same as `NAME.plot_Psi`, but coloured contours of pressure are added. `P_inf` is the far-field pressure for the point where Velocity is zero. `rho` is the gas density used when calculating the local pressure. `P_min` and `P_max` can be used to cap the pressure contours to avoid plotting of $P \to \infty$ close to sources, sinks and vortices.

**Velocity Magnitude** `NAME.plot_magU(magU_max = 100., levels=20)`
Creates a contour plot of velocity magnitude. `magU_max` sets the maximum velocity for the contours. `levels` sets the number of contour lines shown.

**Streamlines** `NAME.plot_Streamlines()`
Creates a plot with fancy looking streamlines. These are based on the $(u, v)$ velocity field, so while these correspond to lines of constant $\Psi$, the difference in streamfunction $\Psi$ between adjacent lines is not constant. Hence streamline separation cannot be related to local velocity.

**Streamlines + Velocity Magnitude** `plot_Streamlines_magU(magU_max = 100., levels=20)`
Combination of the two above functions.

**U-Velocity** `NAME.plot_U(U_min = -100., U_max = 100., levels=20)`
Creates a contour plot of the velocity component in the $x$-direction. `U_min` and `U_max` can be used to cap the maximum velocity that is shown in order to avoid plotting $U \to \infty$ close to sources, sinks and vortices. `levels` sets the number of contour lines shown.

**V-Velocity** `NAME.plot_V(V_min = -100., V_max = 100., levels=20)`
Same as previous function, but for velocity in $y$-direction.

**Pressure** `NAME.plot_P(P_inf = 0., rho=1.225, P_min=-100., P_max=100., levels=20)`
Creates a contour plot of pressure relative to the reference pressure `P_inf` which is defined at a location with zero velocity (This is different to $P_\infty$, which refers to $U_\infty$). `rho` is the fluid density. `P_min` and `P_max` can be used to cap the pressure contours to avoid plotting of $P \to \infty$ close to sources, sinks and vortices. `levels` sets the number of contour lines shown.

**Pressure Coefficient** `NAME.plot_Cp(U_inf = 0., rho=1.225, Cp_min=-5., Cp_max=5., levels=20)`
Creates a contour plot of pressure coefficient defined as $C_p = \frac{P}{\frac{1}{2}\rho U_\infty^2}$. `U_inf` is the free-stream velocity $U_\infty$ `rho` is the fluid density. `Cp_min` and `Cp_max` can be used to cap the $C_p$ contours to avoid plotting of $C_p \to \infty$ close to sources, sinks and vortices. `levels` sets the number of contour lines shown.

## 4.4 Extracting data

In addition to plotting the data it is also possible to evaluate the properties at single points or along lines.

Step: 6

The following code extracts the x-component of velocity, $u$ along the between the points $(-0.5, -0.5)$ and $(-0.5, 0.5)$ and creates a plot of the output data. The results are shown in Fig. 3.

```
# Extract data along lines
T.LinevalU(-0.5,-0.5,-0.5,0.5,plot_flag=1)
T.LinevalPressure(-0.5,-0.5,-0.5,0.5,rho = 1.225, plot_flag=1)

# Make sure plots are displayed on the screen
plt.show()
```

The entire program is executed using the command `python Potential_Flow.py` from the command line.

The possible options for extracting data are described in section 4.4.1.

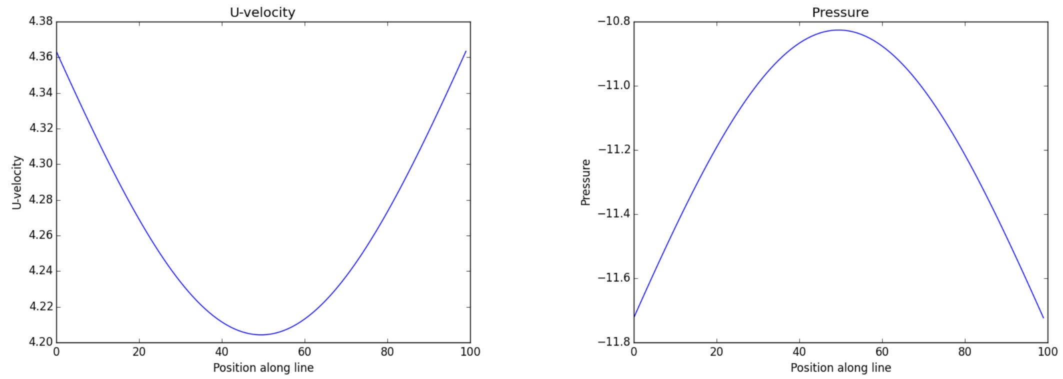
### 4.4.1 Extraction Functions

The following functions extract data from the total flow-field. They must be executed on the PotentialFlow-filed class (e.g. `T`) in the example above. In the description below `NAME` is used as a generic placeholder. These functions must be executed after `NAME.calc([List])`

**Streamfunction** `Psi = NAME.evalP(x,y)`
Returns the total streamfunction magnitude at the point with the coordinates $(x, y)$.

**Velocities** `u, v = NAME.eval(x,y)`
Returns the $x$ and $y$ component of velocity at the point with the coordinates $(x, y)$.

**Pressure** `dP = NAME.evalPressure(x,y,rho)`
Returns the pressure change relative to ambient conditions (zero velocity)




(a) x-component of velocity

(b) Pressure

Figure 3: Flow properties extracted along straight line between the points (-0.5,-0.5) and (-0.5,0.5).

**Line U-Velocity** `UU = NAME.LinevalU(x0,y0,x1,y1,n=100,plot_flag=0)`
Returns the magnitude of velocity in the x-direction, $u$ at $n$ equally spaced points along the line between the two points $(x0, y0)$ and $(x1, y1)$. Setting `plot_flag = 1` will also generate a line graph.

**Line V-Velocity** `VV = NAME.LinevalV(x0,y0,x1,y1,n=100,plot_flag=0)`
Returns the magnitude of velocity in the y-direction, $v$ at $n$ equally spaced points along the line between the two points $(x0, y0)$ and $(x1, y1)$. Setting `plot_flag = 1` will also generate a line graph.

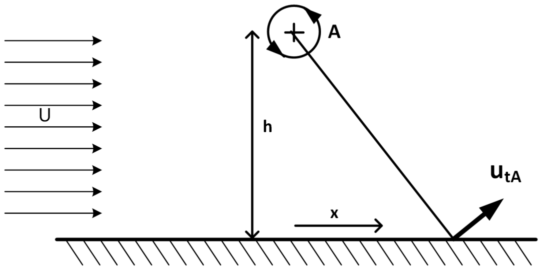
**Line Pressure** `PP = NAME.LinevalPressure(x0,y0,x1,y1,rho,n=100,plot_flag=0)`
Returns the pressure change relative to ambient conditions (zero velocity) at $n$ equally spaced points along the line between the two points $(x0, y0)$ and $(x1, y1)$. Setting `plot_flag = 1` will also generate a line graph.

Instead of exctracting the data from the full flow-field, it is also possible to interrogate a single building block. These functions must be executed on the building block class (e.g. `A1`) in the example above. In the description below `NAME` is used as a generic placeholder.

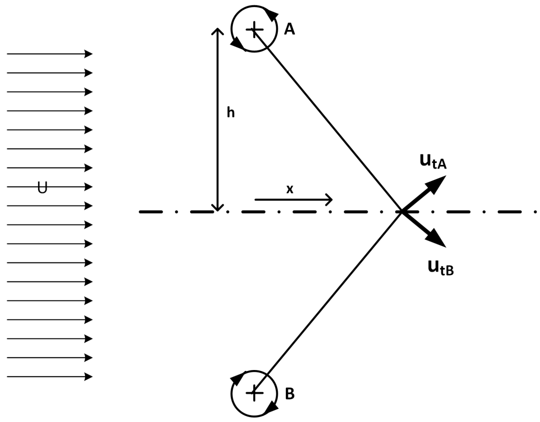
**Velocities** `u, v = NAME.eval(x,y)`
This returns the $x$ and $y$ component of velocity at the point with the coordinates $(x, y)$.

# 5 Example - Vortex near wall

This example shows how `Potential_Flow.py` can be used to analyse the flow field generated by a uniform flow parallel to and a vortex position a distance of 0.5 from the wall. The problem is illustrated in Fig. 4a.




(a) Flow problem




(b) Use aadition of 2nd vortex and use of symetry to create wall

Figure 4: Example case, consisting of uniform flow and a vortex positioned near a wall.

In order to generate the effect of a wall (straight streamline) on can use the principle of symmetry. Thus the problem we will actually solve using Potential Flow theory is the one shown in Fig. 4b, which consists of three building blocks. The Uniform Flow, the Vortex at $(0.0, 0.5)$ and a mirror image (about the x-axis) of the Vortex, located at $(0.0, -0.5)$, which by symmetry generates a straight streamline along the x-axis.

The appropriate code, defining the Uniform Flow, with a strength of 5.0 and vortices with a strength of $\pm 5.0$ is given below. First the building blocks are generated as variables `A1`, `C1`, and `C2`. Then after setting up the flow-field, the list of building blocks `[A1,C1,C2]` is passed to the flow-field solver and evaluated over a $100 \times 100$ grid.

The results from the plotting functions, showing field data and data along the wall, extracted using the `T.LinevalU`, `T.LinevalV` and `T.LinevalPressure` functions are shown in Fig. 5. The obtained velocity in the wall parallel direction equals the analytical solution to the problem, given by

$$\begin{aligned} U_T(x) &= U_\infty + \frac{\Gamma\, h}{\pi\,(x^2 + h^2)} \qquad (8) \\ &= 5.0 + \frac{5.0 \times 0.5}{\pi\left(x^2 + 0.5^2\right)} \\ U_T(0) &= 5.0 + 3.18 = 8.18 \end{aligned}$$

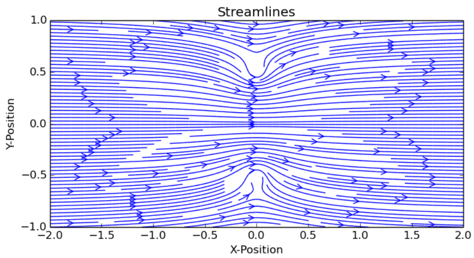
```
if __name__ == "__main__":

    # List of Building Blocks
    # Uniform Flows
```

```
A1 = UniformFlow(5.,0.)
# Vortices
C1 = Vortex(0.0,0.5,-5.)
C2 = Vortex(0.0,-0.5,5.)

# Initialise instance of Plotting Function
T = PlotPotentialFlow()     # create instance of the PotentialFlow-field class
# Set dimensions of Plotting area
T.size(-2.0, 2.0, -1.0 ,1.0)    #(x_min, x_max, y_min, y_max)

# Evaluate PotentialFlow-field over a grid
T.calc([A1,C1,C2],n=100)    # ([List of elements], level of discretisation)

# plot Data over flow-field area
T.plot_Streamlines()     # create Streamline plot.
T.plot_P(P_inf = 0., rho=1.225, P_min=-100., P_max=200.)    # create plot of Press

# extract data at points
# print 'Psi = ', T.evalP(0.,0.)
 print '(u, v) = ', T.eval(0.,0.)
 print 'dP = ', T.evalPressure(0.,0.,rho = 1.225)

# extact data along lines
# lines are defined as x0,y0,x1,y1
T.LinevalU(-2.0,0.0,2.0,0.0,plot_flag=1)
T.LinevalV(-2.0,0.0,2.0,0.0,plot_flag=1)
T.LinevalPressure(-2.0,0.0,2.0,0.0,rho = 1.225, plot_flag=1)

# Make sure plots are displayed on the screen
 plt.show()
```

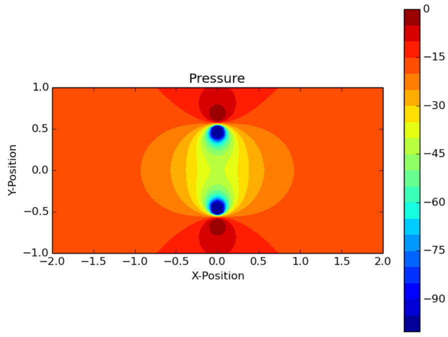
The resulting data is shown in Fig. ??. In addition the following data, corresponding to point extractions is displayed on screen:

```
(u, v) = (8.1830988, 0.0)
dP = -41.0149
```

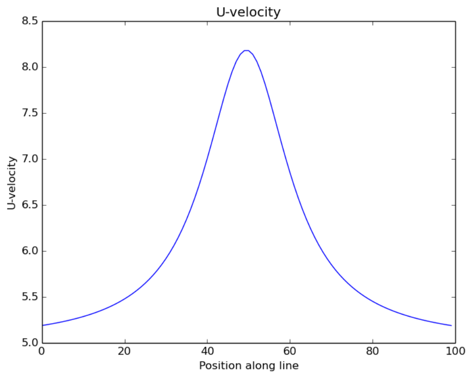
These correspond to the $u$ and $v$-velocity components. Obviously $v = 0$ along the wall and $u = 8.18$, which agrees with the analytical solution for this point. Similar `dP` gives the pressure reduction, calculate as $\Delta P = -\frac{1}{2}\rho U^2 = -41.01$.



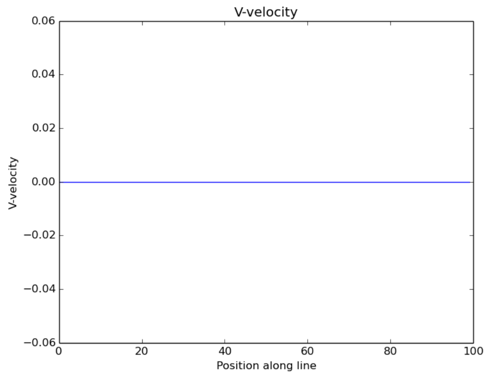

(a) Streamlines



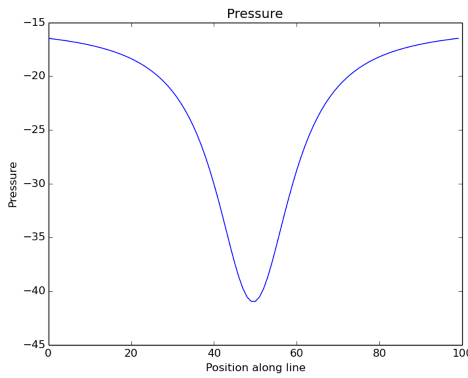

(b) Pressure contours




(c) U-Velocity, extracted along the x-axis




(d) V-velocity, exteacted along the x-axis




(e) Static Pressure, extracted along the x-axis

Figure 5: Flow field and flow properties obtained from a uniform flow with $u$-velocity of 5.0 and a vortex with a strength of $-5.0$ positioned $(0.0, 0.5)$ positioned near a wall running along the x-axis as shown.

# 6 References

## References


[1] CFCFD, *The Compressible Flow Project* `http://cfcfd.mechmining.uq.edu.au` The University of Queensland


# 7 Appendix

## 7.1 Source Code Potential_Flow.py

```
## \Potential_Flow.py
#
"""
Script to create Potential Flow Flow-Fields

Author: Ingo Jahn
Last modified: 27/07/2015
"""

import numpy as np
import matplotlib.pyplot as plt



class PlotPotentialFlow:
    """
    class for PotentiaFlow-fields
    """
    def __init__(self):
        self.size()
    ##
    def size(self, x0=0.0, x1=1.0, y0=0.0, y1 = 1.0):
        self.x0 = x0
        self.x1 = x1
        self.y0 = y0
        self.y1 = y1
    ##
    def calc(self,A,n=100):
        self.A = A
        # create mesh
        xx = self.x0 + np.arange(n) * (self.x1-self.x0) / float(n-1)
        yy = self.y0 + np.arange(n) * (self.y1-self.y0) / float(n-1)
        self.X, self.Y = np.meshgrid(xx,yy)
        self.PSI = np.zeros([n,n])
        self.UU = np.zeros([n,n])
        self.VV = np.zeros([n,n])
        # calculate stream functions and velocities
        for i in range(n):
            x = xx[i]
            for j in range(n):
                y = yy[j]
                psi = 0.
                U = 0.
                V = 0
                for it in range(len(A)):
```

```
                    psi = psi + A[it].evalP(x,y)
                    u,v = A[it].eval(x,y)
                    U = U + u
                    V = V + v
                self.PSI[j,i] = psi
                self.UU[j,i] = U
                self.VV[j,i] = V
    def evalP(self,x,y):
        # calculate Psi at a point
        PSI = 0.
        for it in range(len(self.A)):
            PSI = PSI + self.A[it].evalP(x,y)
        return PSI
    ##
    def eval(self,x,y):
        # calculate U and V at a point
        U = 0.
        V = 0
        for it in range(len(self.A)):
            u,v = self.A[it].eval(x,y)
            U = U + u
            V = V + v
        return U, V
    ##
    def evalPressure(self,x,y,rho):
        # calculate pressure reduction
        u,v = self.eval(x,y)
        Umag2 = u**2 + v**2
        dP = - 0.5 * rho * Umag2
        return dP
    ##
    def LinevalU(self,x0,y0,x1,y1,n=100,plot_flag=0):
        # calculate u-velocity at N points linearly spaced between point 0 and 1
        xx = x0 + np.arange(n) * (x1-x0) / float(n-1)
        yy = y0 + np.arange(n) * (y1-y0) / float(n-1)
        UU = np.zeros(n)
        for i in range(n):
            u,v = self.eval(xx[i],yy[i])
            UU[i] = u
        if plot_flag == 1:
            plt.figure()
            plt.plot(UU)
            plt.title('U-velocity')
            plt.xlabel('Position along line')
            plt.ylabel('U-velocity')
        return UU
    ##
    def LinevalV(self,x0,y0,x1,y1,n=100,plot_flag=0):
        # calculate u-velocity at N points linearly spaced between point 0 and 1
        xx = x0 + np.arange(n) * (x1-x0) / float(n-1)
        yy = y0 + np.arange(n) * (y1-y0) / float(n-1)
        VV = np.zeros(n)
        for i in range(n):
            u,v = self.eval(xx[i],yy[i])
            VV[i] = v
        if plot_flag == 1:
            plt.figure()
            plt.plot(VV)
            plt.title('V-velocity')
            plt.xlabel('Position along line')
            plt.ylabel('V-velocity')
        return VV
```

```
    ##
    def LinevalPressure(self,x0,y0,x1,y1,rho,n=100,plot_flag=0):
        # calculate u-velocity at N points linearly spaced between point 0 and 1
        xx = x0 + np.arange(n) * (x1-x0) / float(n-1)
        yy = y0 + np.arange(n) * (y1-y0) / float(n-1)
        PP = np.zeros(n)
        for i in range(n):
            u,v = self.eval(xx[i],yy[i])
            PP[i] = - 0.5 * rho * (v**2 + u**2)
        if plot_flag == 1:
            plt.figure()
            plt.plot(PP)
            plt.title('Pressure')
            plt.xlabel('Position along line')
            plt.ylabel('Pressure')
        return PP
    ##
    def plot_Streamlines(self):
        plt.figure()
        plt.streamplot(self.X,self.Y,self.UU,self.VV, density = 2, linewidth = 1, arrowsize=2, arrowstyle='->')
        #plt.scatter(self.x0,self.y0,color='#CD2305', s=80, marker='o',linewidth=0)
        plt.title('Streamlines (not potentials)')
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0,self.x1])
        plt.gca().set_ylim([self.y0,self.y1])
    ##
    def plot_Streamlines_magU(self,magU_max = 100,levels=20):
        plt.figure()
        magU = (self.VV**2 + self.UU**2)**0.5
        magU[magU>magU_max] = magU_max
        CS = plt.contourf(self.X, self.Y, magU,levels)
        plt.colorbar(CS)
        plt.streamplot(self.X,self.Y,self.UU,self.VV, density = 2, linewidth = 1, arrowsize=2, arrowstyle='->',color='k')
        plt.title('Streamlines (not potentials) and Velocity Magnitude')
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0,self.x1])
        plt.gca().set_ylim([self.y0,self.y1])
    ##
    def plot_Psi_magU(self,magU_max = 100,levels=20):
        plt.figure()
        magU = (self.VV**2 + self.UU**2)**0.5
        magU[magU>magU_max] = magU_max
        CS = plt.contourf(self.X, self.Y, magU,levels)
        plt.colorbar(CS)
        CS2 = plt.contour(self.X, self.Y, self.PSI,levels,colors='k')
        plt.clabel(CS2,fontsize=9,inline=1)
        plt.title('Streamfunctions PSI and Velocity Magnitude')
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0,self.x1])
        plt.gca().set_ylim([self.y0,self.y1])
    ##
    def plot_U(self,U_min = -100., U_max = 100., levels=20):
        U = self.UU
```

```
        U[U<U_min] = U_min
        U[U>U_max] = U_max
        self.plot_cf(U, levels=levels, label="U-velocity")
    ##
    def plot_V(self, V_min = -100., V_max = 100., levels=20):
        V = self.VV
        V[V<V_min] = V_min
        V[V>V_max] = V_max
        self.plot_cf(V, levels=levels, label="V-velocity")
    ##
    def plot_magU(self, magU_max = 100, levels=20):
        magU = (self.VV**2 + self.UU**2)**0.5
        magU[magU>magU_max] = magU_max
        self.plot_cf((self.VV**2 + self.UU**2)**0.5, levels=levels, label="Velocity Magnitude")
    ##
    def plot_cf(self, Z, levels=20, label="Label"):
        plt.figure()
        CS = plt.contourf(self.X, self.Y, Z, levels)
        plt.colorbar(CS)
        plt.title(label)
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.legend
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0, self.x1])
        plt.gca().set_ylim([self.y0, self.y1])
    ##
    def plot_P(self, P_inf = 0., rho=1.225, P_min=-100., P_max=100., levels=20):
        # limit pressure to
        P = P_inf - 0.5 * rho * (self.VV**2 + self.UU**2)
        P[P<P_min] = P_min
        P[P>P_max] = P_max
        self.plot_cf(P, levels=levels, label="Pressure")
    ##
    def plot_Psi_P(self, P_inf = 0., rho=1.225, P_min = -100., P_max = 100., levels=20):
        plt.figure()
        P = P_inf - 0.5 * rho * (self.VV**2 + self.UU**2)
        P[P<P_min] = P_min
        P[P>P_max] = P_max
        CS = plt.contourf(self.X, self.Y, P, levels)
        plt.colorbar(CS)
        CS2 = plt.contour(self.X, self.Y, self.PSI, 2*levels, colors='k')
        plt.clabel(CS2, fontsize=9, inline=1)
        plt.title('Streamfunctions PSI and Pressure')
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0, self.x1])
        plt.gca().set_ylim([self.y0, self.y1])
    ##
    def plot_Cp(self, U_inf = 0., rho=1.225, Cp_min = -5., Cp_max = 5., levels=20):
        if float(U_inf) == 0.:
            print "For case with U_inf = 0., Cp becomes infinite everywhere"
        else:
            # Limit CP to account for localised high velocities
            Cp = (0.5* rho*U_inf**2 - 0.5* rho * (self.VV**2 + self.UU**2)) / (0.5 * rho * U_inf**2)
            Cp[Cp < Cp_min] = Cp_min
            Cp[Cp > Cp_max] = Cp_max
```

```
            self.plot_cf(Cp,levels=levels,label="Pressure Coefficient - Cp (Note
                limited to +/-5.)")
    ##
    def plot_Psi_contours(self, levels=20):
        # Plot stream function Psi as coloured contours
        label="Streamfunction PSI Contours"
        plt.figure()
        CS = plt.contourf(self.X, self.Y, self.PSI, levels, cmap='hsv')
        # set graph details
        plt.colorbar(CS)
        plt.title(label)
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.legend
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0,self.x1])
        plt.gca().set_ylim([self.y0,self.y1])
    ##
    def plot_Psi(self, levels=20):
        # Plot stream function Psi
        label="Streamfunction PSI"
        plt.figure()
        CS = plt.contour(self.X, self.Y, self.PSI,levels,colors='k')
        plt.clabel(CS,fontsize=9,inline=1)
        # set graph details
        plt.title(label)
        plt.xlabel('X-Position')
        plt.ylabel('Y-Position')
        plt.legend
        plt.gca().set_aspect('equal')
        plt.gca().set_xlim([self.x0,self.x1])
        plt.gca().set_ylim([self.y0,self.y1])


## Definition of classes used as Building Blocks
class UniformFlow:
    """
    class that creates a uniform flow field for potential flow
    UniformFlow(u,v)
    u  -  x-component of velocity
    v  -  y-component of velocity
    """
    def __init__(self,u,v):
        self.u = u
        self.v = v
    ##
    def evalP(self,x,y):
        P = self.u*y - self.v*x
        return P
    ##
    def eval(self,x,y):
        u = self.u
        v = self.v
        return u,v

class Source:
    """
    class that creates a source for potential flow.
    Source(x0,y0,m)
    x0  -  x-position of Source
    y0  -  y-position of Source
    m   -  total flux generated by source (for sink set -ve)
```

```python
    """
    def __init__(self,x0,y0,m):
        self.x0 = x0
        self.y0 = y0
        self.m = m
    ##
    def evalP(self,x,y):
        theta = np.arctan2(y-self.y0,x-self.x0)
        P = theta * self.m /(2*np.pi)
        return P
    ##
    def eval(self,x,y):
        r = ( (x-self.x0)**2 + (y-self.y0)**2 )**0.5
        u = self.m / (2*np.pi) * (x - self.x0) / (r**2)
        v = self.m / (2*np.pi) * (y - self.y0) / (r**2)
        return u,v

class Vortex:
    """
    class that creates an irrotational vortex for potential flow.
    Vortex(x0,y0,K)
    x0 - x-position of Vortex core
    y0 - y-position of Vortex core
    K  - Strength of Vortex
    """
    def __init__(self,x0,y0,K):
        self.x0 = x0
        self.y0 = y0
        self.K = K
    ##
    def evalP(self,x,y):
        r = ( (x-self.x0)**2 + (y-self.y0)**2 )**0.5
        P = - self.K * np.log(r)
        return P
    ##
    def eval(self,x,y):
        r = ( (x-self.x0)**2 + (y-self.y0)**2 )**0.5
        u = self.K / (2*np.pi) * (y - self.y0) / (r**2)
        v = - self.K / (2*np.pi) * (x - self.x0) / (r**2)
        return u,v

class Doublet:
    """
    class that creates a doublet.
    If combined with uniform flow of veloicty U_inf in the +x direction, this creates the flow around a cylidner.
    Doublet(x0,y0,a,U_inf)
    x0 - x-position of Vortex core
    y0 - y-position of Vortex core
    a  - radius of cylinder generated if superimposed to Uniform Flow
    U-inf - Strength of uniform flow
    """
    def __init__(self,x0,y0,a,U_inf):
        self.x0 = x0
        self.y0 = y0
        self.a = a
        self.U_inf = U_inf
    def evalP(self,x,y):
        P = self.U_inf * (y-self.y0) * ( - self.a**2 / ((y-self.y0)**2 + (x-self.x0)**2) )
        # set to zero inside circle
        #if ((y-self.y0)**2 + (x-self.x0)**2) < self.a**2:
```

```
        #     P = np.nan
        return P
    ##
    def eval(self,x,y):
        u = self.U_inf * self.a**2 * - ((x-self.x0)**2 - (y-self.y0)**2) / ( ((x-self.x0)**2 + (y-self.y0)**2)**2 )
        v = self.U_inf * self.a**2 * -2. * (x-self.x0) * (y-self.y0) / ( ((x-self.x0)**2 + (y-self.y0)**2)**2 )
        #if ((y-self.y0)**2 + (x-self.x0)**2) < self.a**2:
        #     u = np.nan
        #     v = np.nan
        return u,v

class User_Defined:
    """
    Special Userdefined building block (flow through 90 degree corner)
    User_Defined(x0,y0,A)
    x0  -  x-position of corner
    y0  -  y-position of corner
    A   -  Strength of Flow
    """
    def __init__(self,x0,y0,A):
        self.x0 = x0
        self.y0 = y0
        self.A = A
    ##
    def evalP(self,x,y):
        P = self.A * (x-self.x0) * (y-self.y0)
        return P
    ##
    def eval(self,x,y):
        u = self.A * (x-self.x0)
        v = - self.A * (y-self.y0)
        return u,v

class Name:
    """
    Template for user generated building blocks
    Name(x0,y0,Var1,Var2,Var3)
    x0  -  x-position
    y0  -  y-position
    Var1  -  Variable1
    Var2  -  Variable2
    Var3  -  Variable3
    """
    def __init__(self,x0,y0,Var1,Var2,Var3):
        self.x0 = x0
        self.y0 = y0
        self.Var1 = Var3
        self.Var2 = Var2
        self.Var3 = Var1
    ##
    def evalP(self,x,y):
        ## Function for stream function goes here
        P = 0
        return P
    ##
    def eval(self,x,y):
        ## Functions for u and v velocity go here. I.e. differentiate stream function with respect to x and y
        u = 0
        v = 0
```

```python
        return u,v



## Main section of the code, executed if running the file
if __name__ == "__main__":
    ###########################
    # List of Building Blocks #
    ###########################
    # Uniform Flows
    A1 = UniformFlow(5.,0.)
    A2 = UniformFlow(0.,0.)
    # Vortices
    C1 = Vortex(0.0,0.0,5.)
    C2 = Vortex(0.0,-0.5,0.5)
    C3 = Vortex(0.0,0.0,10.0)
    # Sources
    D1 = Source(0.5,0., 10.)
    D2 = Source(0.1,0.,-100.)
    D3 = Source(0.1,0.,0.24)
    # User Defined functions
    U1 = User_Defined(0.,0.,5)
    DD = Doublet(0.,0.,0.1,5.)

    # ##########################################
    # Initialise instance of Plotting Function #
    ############################################
    T = PlotPotentialFlow()     # create instance of the PotentialFlow-field class
    # Set dimensions of Plotting area
    T.size(-2.0, 2.0, -1.0 ,1.0)    #(x_min, x_max, y_min, y_max)

    ###########################################
    # Assemble Building Blocks and Calculate #
    ###########################################
    T.calc([A1,D1],n=100)    # ([List of elements], level of discretisation)

    ###################################
    # plot Data over flow-field area #
    ###################################
    T.plot_Psi(levels = 40)                         # plot lines of constatant Psi
    #T.plot_Psi_contours(levels = 20)               # plot contours of Psi
    T.plot_Psi_magU(magU_max=10., levels=40)        # plot lines of Psi + velocity contours
    T.plot_Psi_P(P_inf=0., rho=1.225, P_min=-30., P_max=30., levels=20) # plot lines of Psi + pressure contours
    #T.plot_magU(magU_max = 10., levels = 20)       # plot contours of Velocity magnitude
    #T.plot_Streamlines()                           # plot Streamline plot.
    #T.plot_Streamlines_magU(magU_max = 10.,levels = 20)    # plot Streamlines + velocity magnitude
    #T.plot_U(U_min = -10., U_max = 10., levels = 20)       # plot of U-velocity contours
    #T.plot_V(V_min = -10., V_max = 10., levels = 20)       # plot of V-velocity contours
    #T.plot_P(P_inf = 0., rho=1.225, P_min=-100., P_max=200.)   # plot Pressure contours
    #T.plot_Cp(U_inf = 5., rho=1.225, Cp_min = -5., Cp_max = 5., levels=20) # plot contours of Pressure coefficient

    ##########################
    # extract data at points #
    ##########################
```

```
    #print 'Psi = ', T.evalP(0.,0.)
    #print '(u, v) = ', T.eval(0.,0.)
    #print 'dP = ', T.evalPressure(0.,0.,rho = 1.225)

    ############################
    # extract data along lines #
    ############################
    # lines are defined as x0,y0,x1,y1
    #T.LinevalU(-0.5,-0.5,-0.5,0.5,plot_flag=1)
    #T.LinevalV(-0.5,-0.5,-0.5,0.5,plot_flag=1)
    #T.LinevalPressure(-0.5,-0.5,-0.5,0.5,rho = 1.225, plot_flag=1)


    # Make sure plots are displayed on the screen
    plt.show()
```