ME106 Project1 Report

Rui Wang
3033461836

Contents

1 Part 1: Streamlines

1.1 Problem Statement

The problem consists of two parts:

- Given an array of velocity field, plot it with MATLAB `quiver()` function.
- Draw ten streamlines on the plot.

1.2 Solution

The array is given as a 63×63 matrix, however, we do not want to hard code it as we wish to apply this algorithm to any dimensions of matrices as well. Plotting the velocity field is straight-forward, and streamlines are obtained by moving along the velocity field with a constant time('snapshot').

1.3 Code and Plot for Velocity Field

First, we plot the velocity field with `quiver()`. Note that the matrix must be scaled to $[0, 1]$ before plot, so we make use of `meshgrid()`.

Listing 1: Commands to plot a velocity field.

```
load('vel.mat');
x = linspace(0, 1, 63);
y = linspace(0, 1, 63);
[x, y] = meshgrid(x, y);
quiver(x, y, udata, vdata);
```

Figure 1 is the plot for the velocity field.

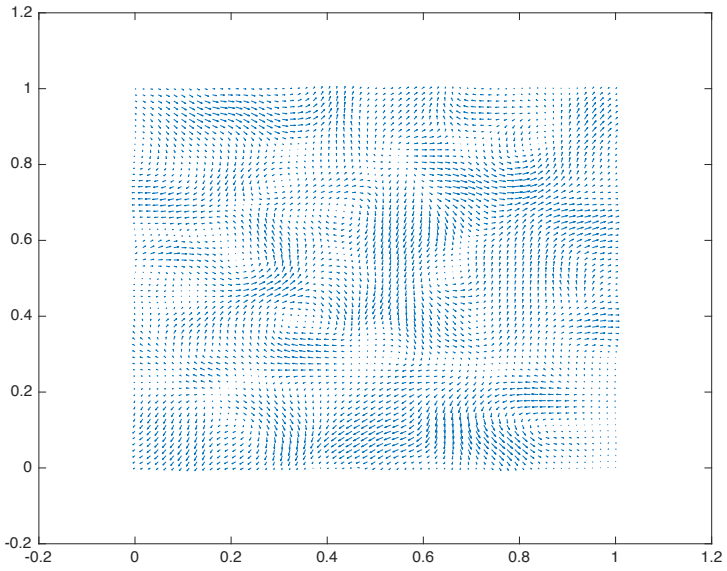

Figure 1: The original velocity field.

1.4 Code and Plot for Streamline

Below is the function to plot a streamline. `xcoord` and `ycoord` are references for x and y axis, and can either be a column or row vector. The entire area is presumed to be a rectangle. `x0` and `y0` are column vectors, namely the x, y axis of a list of starting point of the streamlines to be plotted.

Listing 2: streamlinePlotter.m

```
function streamlinePlotter(xcoord, ycoord, udata, vdata, x0, y0)
% hold off % prevent covering last figure
ds = 0.001; % set the step to be 0.001
n = max(size(xcoord));
  for i = 1 : size(x0, 1)
  xNew = x0(i); yNew = y0(i); % set the initial location
  x = []; y = [];
  while exceed(xNew, yNew, xcoord, ycoord) == 0 && visited(xNew,
     yNew, x, y) == 0
    % if reaches edge or reaches a point for the second time, stop
       plotting
    x = [x, xNew]; y = [y, yNew];
    xint = (xNew-xcoord(1)) / (xcoord(end)-xcoord(1)+0.0) * (n-1) +
       1; % scale for interpolation
    yint = (yNew-ycoord(1)) / (ycoord(end)-ycoord(1)+0.0) * (n-1) +
       1;
    u = interp2(udata, xint, yint);
    % scale (xNew, yNew) up to interpolate
    v = interp2(vdata, xint, yint);
    xNew = xNew + ds * u;
    yNew = yNew + ds * v;
  end
  plot(x, y);
  hold on % draw on the same figure
  end
end

function check = exceed(x, y, xcoord, ycoord) % check if point
   exceeds the graph boundary
  check = false;
  if (x > xcoord(end)) || (x < xcoord(1)) || (y < ycoord(1)) || (y >
      ycoord(end))
      check = true;
  end
end

function check = visited(xNew, yNew, x, y) % check if a point is
   visited before
  check = false;
  if ismember([xNew; yNew], [x; y])
      check = true;
  end
end
```

We then obtain our streamline with the following commands:(using `ginput()` to obtain ten appropriate starting points on Figure 1, so we keep the previous figure open). With the following commands, we can plot our streamline:

Listing 3: Plot the streamline with my function

```
x = linspace(0, 1, 63);
y = linspace(0, 1, 63);
[xst, yst] = ginput(10);
hold on
streamlinePlotter(x, y, udata, vdata, xst, yst);
```

The basic idea is to `walk` out a streamline. As for a streamline, we have

$$\frac{dx}{u} = \frac{dy}{v} = ds$$

thus we obtain

$$dx = uds, dy = vds$$

dx and dy are the distances of our each step.

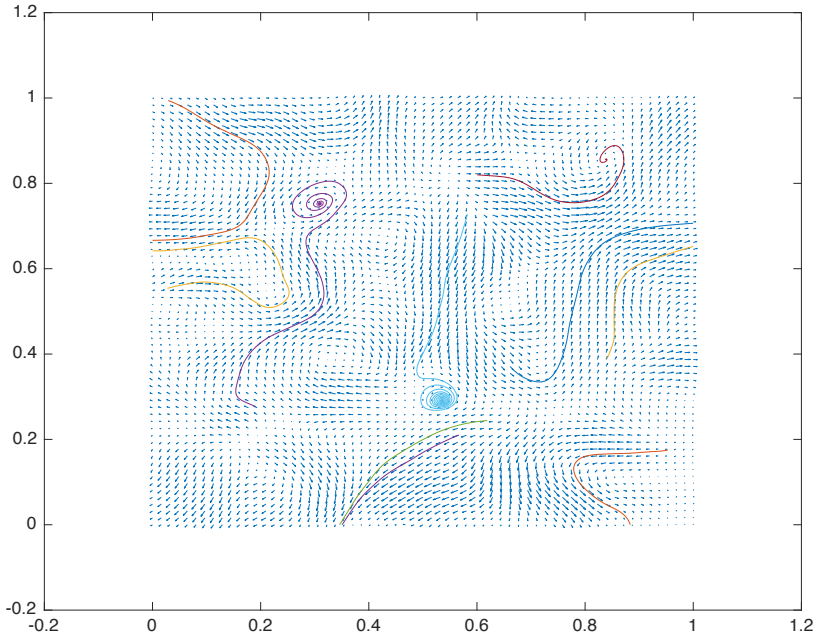

Figure 2: Streamlines plotted by my function.

We can compare it with MATLAB's `streamline()` function, the result of which is on Figure 3. We can observe that the curves of two methods coincide, justifying my implementation. Note that on Figure 3, the new curve is directly plotted upon the previous one. The dark blue lines are plotted by MATLAB's `streamline()`. We can see that there is a little difference(especially on the vortex in the middle), this is because we are using a fixed step of $ds = 0.001$, which can yield inaccurate estimation at certain locations where the curvature is very large.

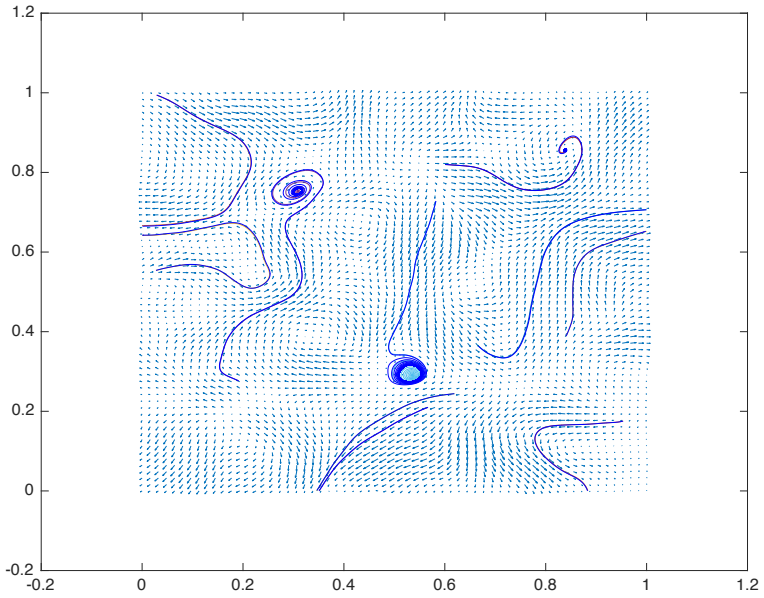

Figure 3: Streamlines plotted by MATLAB.

2 Part 2: Flow around a Moving Cylinder

2.1 Problem Satatement

The flow velocity field around a cylinder is given. Since $a = 1$ and $U = 1$ throughout the problem, we directly plug in these two constants and get the following expression:

$$u = \frac{(x+t)^2 - y^2}{((x+t)^2 + y^2)^2}$$

$$v = \frac{2(x+t)y}{((x+t)^2 + y^2)^2}$$

We are required to

- plot the velocity field using `contourf()` when $t = 0$.
- Plot 8 streamlines with given starting point.
- Plot a pathline.

2.2 Solution Code and Result for Velocity Field

With a first trial, we will see that the velocity goes to infinity very quickly at around the origin. This makes the contour extremely ugly, so we sort to draw the contour not based on a uniform distance in velocity, but on a gradual gradient.

One thing to note is that we need to enlarge each step from 0.001 to maybe 0.01 in this implementation to allow for a faster calculation. Otherwise we will have to wait for a very long time before the plot can be finished.

Below is the code for the first two requirements of this problem.

Listing 4: Velocity contour plotting and streamline at given points.

```
xst = [-2 2 -0.75 -0.75 -0.5 -0.5 -0.25 -0.25]';
yst = [0 0 1 -1 1 -1 1 -1]';
x = [-4:0.01:4]; y = x; n = size(x, 2);
x = repmat(x, n, 1); y = repmat(y', 1, n);
t = 0;
u = ((x+t).^2-y.^2)./((x+t).^2+y.^2).^2; v = 2*(x+t).*y./((x+t).^2+y
    .^2).^2; vel = (v.^2+u.^2).^0.5;
contourf(x, y, vel, [0.03, 0.05, 0.1, 0.2, 0.4, 1.0, 2, 4, 8, 20])
hold on
streamlinePlotter(x(1, :), y(:, 1), u, v, xst, yst)
```

This yields

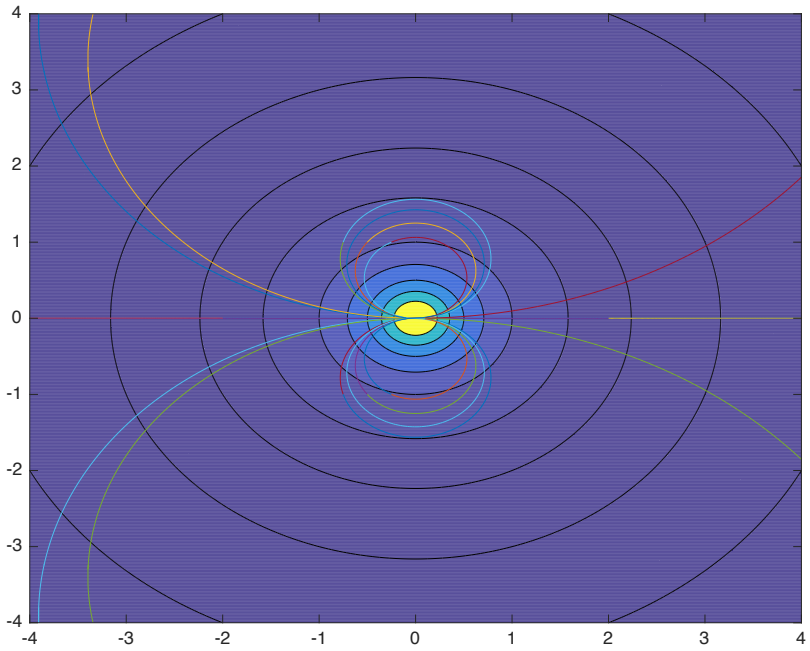

Figure 4: Velocity Contour and Streamlines

In Figure 4, black lines indicate the contour, while colored lines are those streamlines.

2.3 Code and Result for Pathline

For the pathline problem, we use basically the same method as problem Part 1, except for that we include time t as a variable in this numerical simulation. One problem is that the velocity of the bug is expressed as u_r and u_θ, while in reality if we want to calculate the next position for the bug we need the x, y coordinates. So we will need to convert from polar to cartesian.

$$u_x = u_r cos\theta - u_\theta sin\theta$$
$$u_y = u_r sin\theta + u_\theta cos\theta$$

Hereafter we can update the bug's position by $x_t = x_{t-dt} + u_x dt$ and $y_t = y_{t-dt} + u_y dt$.

```
t = 0;
dt = 0.2; % as required in the problem statement
x0 = -1.1; y0 = 0.02; % init position
x = zeros(1, 500001); y = zeros(1, 500001);
idx = 1; % step index
while t <= 100000
    x(idx) = x0;
    y(idx) = y0;
    R = sqrt((x0+t)^2 + y0^2);
    c = (x0+t)/R;
    s = y0/R;
    cosine = x0/sqrt(x0^2+y0^2); % polar-cartesian angle
    sine = y0/sqrt(x0^2+y0^2);
    u_r = -1/R^2*c; % u_r
    u_t = -1/R^2*s; % u_theta
    u = u_r*cosine-u_t*sine; % velocity in cartesian space
    v = u_r*sine+u_t*cosine;
    x0 = x0 + dt*u;
    y0 = y0 + dt*v; % update position
    idx = idx + 1;
    t = t + dt;
end
plot(x(1:idx-1), y(1:idx-1))
set(gca,'XLim',[-4 4],'YLim',[-4 4])
```

The result of the pathline is a bit boring, as shown in Figure 5.

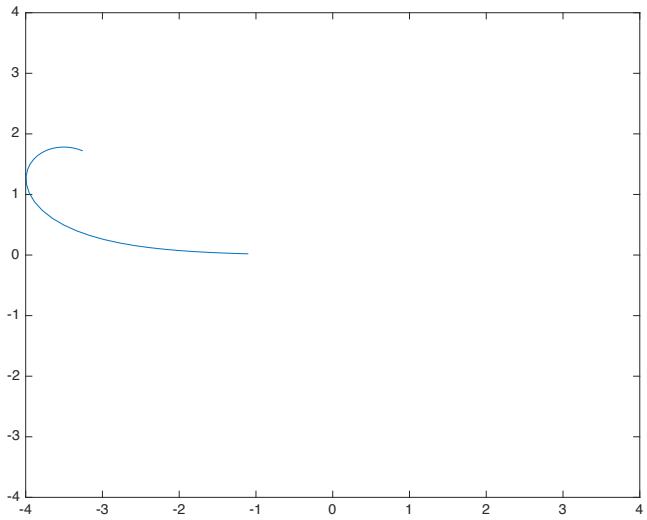

Figure 5: Pathline of a Bug

However, the size of the step may affect the plotting result, so the pathline is not rigorous.