McGill University

MECH 309

Project 1 in Numerical Methods

Bao, Weijie 260360432
Harizaj, Dritan 260426327
3/26/2012

Table of Contents

1 Basins of Attraction

In part 1 of this project, consider this system of two non-linear equations:

$$\begin{cases} x^2+y^2-4=0 \\ \dfrac{(x-1)^2}{9}+\dfrac{(y-1)^2}{4}-1=0 \end{cases} \quad (1)$$

The above non-linear system is analyzed and solved in the following three ways: fixed-point iteration, Newton's method and Broyden's method.

1.1 Fixed point of iteration

Use fixed –point iteration to do the find the two solutions (x_1,y_1) and (x_2,y_2) to the system up to 8 digits. Since this function has two solutions, there are two while loops to find the solutions. Solving the equations using fixed-point iteration, we have to rearrange the equations to isolate x and y in the following format:

$$\begin{cases} x=\pm\sqrt{4-y^2} \\ y=1\pm\sqrt{4[1-\dfrac{(x-1)^2}{9}]} \end{cases} \quad (2)$$

As shown in the above system, there are two equations separately corresponding to x and y. There should be four ways to iterate and find out the solutions theoretically. However, only two forms could be used to solve the system. The details of how to select the set of (x,y) are shown in Section 1.1.1 below.

1.1.1 Prove existence of the fixed-points

This system of 2 nonlinear equations in 2 unknowns can be represented by defining a function **F** mapping $\mathbb{R}^2$ into $\mathbb{R}^2$ as:

$$\mathbf{F}(\mathbf{x})=\mathbf{F}(x_1,x_2)=\begin{bmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2)\end{bmatrix}=\begin{bmatrix} {x_1}^2+{x_2}^2-4 \\ \dfrac{(x_1-1)^2}{9}+\dfrac{(x_2-1)^2}{4}-1\end{bmatrix}$$

There are four forms of **G** (**x**):

Let $\mathbf{G_1}$: $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ be defined by $\mathbf{G_1}$ (**x**) =$(\mathrm{g}_1(\mathbf{x}),\mathrm{g}_2(\mathbf{x}))^{\mathrm{t}}$ as shown in system 2, where:

$$\mathrm{g}_1(x_1,x_2)=x_1=-\sqrt{4-{x_2}^2}, -2\le \mathrm{g}_1\le 0$$

$$\mathrm{g}_2(x_1,x_2)=x_2=1-\sqrt{4\left[1-\frac{(x_1-1)^2}{9}\right]}, -1\le \mathrm{g}_2\le 1$$

Let $\mathrm{D}=\{(x_1,x_2)^{\mathrm{t}}|-2\le x_i\le 2 \text{ for i} = 1{,}2\}$

$$\mathbf{G_1}\ (\mathbf{x})=\begin{bmatrix} -\sqrt{4-{x_2}^2} \\ 1-\sqrt{4\left[1-\dfrac{(x_1-1)^2}{9}\right]}\end{bmatrix}$$

Because $\mathbf{G_1}$ is continuous from D $\subset\mathbb{R}^2$ into $\mathbb{R}^2$ and $\mathbf{G_1}$ $(\mathbf{x}) \in$ D, where $\mathbf{x} \in$ D, $\mathbf{G_1}$ has a fixed point in D.

Another Form of G is shown below:
Let $\mathbf{G_2}$: $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ be defined by $\mathbf{G_2}$ $(\mathbf{x})$ =$(\mathrm{g}_1(\mathbf{x}), \mathrm{g}_2(\mathbf{x}))^{\mathrm{t}}$, where:

$$\mathrm{g}_1(x_1, x_2) = x_1 = \sqrt{4 - {x_2}^2}, 0 \le \mathrm{g}_1 \le 2$$

$$\mathrm{g}_2(x_1, x_2) = x_2 = 1 - \sqrt{4\left[1 - \frac{(x_1 - 1)^2}{9}\right]}, -1 \le \mathrm{g}_2 \le 1$$

Let $\mathrm{D} = \{(x_1, x_2)^{\mathrm{t}} | -2 \le x_i \le 2 \text{ for i} = 1{,}2\}$

$$\mathbf{G_2}\ (\mathbf{x}) = \begin{bmatrix} \sqrt{4 - {x_2}^2} \\ 1 - \sqrt{4\left[1 - \frac{(x_1 - 1)^2}{9}\right]} \end{bmatrix}$$

Because $\mathbf{G_2}$ is continuous from D $\subset\mathbb{R}^2$ into $\mathbb{R}^2$ and $\mathbf{G_2}$ $(\mathbf{x})$ $\in$D, where x$\in$D, $\mathbf{G_2}$ has a fixed point in D as well.

There are two other forms of $\mathbf{G}(\mathbf{x})$:

$$\mathbf{G_3}\ (\mathbf{x}) = \begin{bmatrix} -\sqrt{4 - {x_2}^2} \\ 1 + \sqrt{4\left[1 - \frac{(x_1 - 1)^2}{9}\right]} \end{bmatrix}$$

$$\textit{and}\ \mathbf{G_4}\ (\mathbf{x}) = \begin{bmatrix} \sqrt{4 - {x_2}^2} \\ 1 + \sqrt{4\left[1 - \frac{(x_1 - 1)^2}{9}\right]} \end{bmatrix}$$

where $\mathrm{g}_2(x_1, x_2) = x_2 = 1 - \sqrt{4\left[1 - \frac{(x_1 - 1)^2}{9}\right]}, 1 \le \mathrm{g}_2 \le 3 \notin D$

$$\mathrm{D} = \{(x_1, x_2)^{\mathrm{t}} | -2 \le x_i \le 2 \text{ for i} = 1{,}2\}$$

So $\mathbf{G_3}$ $(\mathbf{x})$ and $\mathbf{G_4}$ $(\mathbf{x})$ do not have a fixed point in $\mathbb{R}^2$, because $\mathbf{G_3}$ $(\mathbf{x})$ and $\mathbf{G_4}$ $(\mathbf{x})$ is not contained in D for x in D.

1.1.2 Uniqueness

According to a theorem in the text book:
"Let $\mathrm{D} = \{(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_\mathrm{n})^t \mid a_i \le \mathrm{x}_\mathrm{i} \le b_i, for\ each\ i = 1{,}2, \ldots, n\}$ for some collection of constants $\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_\mathrm{n}$ and $\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_\mathrm{n}$. Suppose $\mathbf{G}$ is a continuous function from $\mathrm{D} \subset \mathbb{R}^n$ into $\mathbb{R}^n$ with the property that $\mathbf{G}$ $(\mathbf{x})$ $\in D$ whenever $\mathbf{x} \in D$. Then $\mathbf{G}$ has a fixed point in D.

Suppose that all the component functions of G have continuous partial derivatives and a constant K < 1 exists with

$$\left|\frac{\partial g_i(\boldsymbol{x})}{\partial x_j}\right| \le \frac{K}{n}, where\ \boldsymbol{x} \in D,$$

for each j=1,2,…,n and each component function $\mathrm{g_i}$."

Finding bounds for the partial derivatives on D gives:

$$\left|\frac{\partial g_1}{\partial x_1}\right| = 0, \qquad \left|\frac{\partial g_2}{\partial x_1}\right| = 0,$$

as well as

$$\left|\frac{\partial g_1}{\partial x_2}\right| = \left|\frac{x_2}{\sqrt{4-{x_2}^2}}\right|, where - 2 < x_2 < 2$$

$$\left|\frac{\partial g_2}{\partial x_1}\right| = \left|\frac{2(x_1-1)}{3\sqrt{-{x_1}^2-2x_1+8}}\right|, where - 2 < x_1 < 4$$

Since the maximum K is larger than 1, it fails to use the theorem we learned in class to prove the uniqueness of the system.

1.1.3 MATLAB Algorithm

Arbitrarily select (2,0) as our initial guess.

```
function [] = fixed (x0,y0)
% Tolerance and format are set.
% Some preliminary variables are defined as well.
format long;
tol=10e-8;
LeNorme=1;
x0=2;
y0=0;
x_input=x0;
y_input=y0;
X0=[x_input;y_input];
k=0;
% Implemented the Fixed point algorithm discussed in class.
% To find the fixed point function, the equations were
% rearranged by hand to isolate x and y.
%
% 1st Iteration
% +sqrt and -sqrt
while (LeNorme > tol)
    k=k+1;
    X1=[sqrt(4-y0*y0);1-sqrt(4-(4/9 *(x0-1)^2))];
    deltaX=X1-X0;
    LeNorme=norm(deltaX);
    X0=X1;
    x0=X1(1);
    y0=X1(2);
    if (k==1000)
        fprintf('Script unresponsive; too many iterations.');
        break;
    end
end
S1=single(X1)
k1=k
%
```

```
% 2nd Iteration
% -sqrt and -sqrt
LeNorme=1;
X0=[x_input;y_input];
k=0;
%
while (LeNorme > tol)
    k=k+1;
    X1=[-sqrt(4-y0*y0);1-sqrt(4-(4/9 *(x0-1)^2))];
    deltaX=X1-X0;
    LeNorme=norm(deltaX);
    X0=X1;
    x0=X1(1);
    y0=X1(2);
    if (k==1000)
        fprintf('Script unresponsive; too many iterations.');
        break;
    end
end
S2=single(X1)
k2=k
%
return;
```

Running the code above we get two sets of solutions:

$$S1 = \begin{cases} x = -1.9229734 \\ y = 0.5497031 \end{cases}$$

k1 =158;
It takes 158 iterations to get this solution.

$$S2 = \begin{cases} x = 1.7689439 \\ y = -0.9331868 \end{cases}$$

k2 =16;
It takes 16 iterations to get this solution.

1.2 Newton's Method

In this section, Newton's method is introduced to find solutions to the system. Basins of attraction of solutions (x1,y1) and (x2,y2)is described by scanning initial input (x0, y0). Using a color scale can illustrate how fast Newton's method converges to one of the solutions.

1.2.1 MATLAB Algorithm

```
function [] = newton2()
    % Define a function that carries out
    % Gaussian Elimination with Partial Pivoting
    % and Back Substitution
    %
    function [F] = LeFonction(u0,v0)
    fu0=u0^2+v0^2-4;
    fv0=((u0-1)^2)/9 + ((v0-1)^2)/4 -1;
    F=[fu0;fv0];
    end
    %
    % Define a function that carries out
    % Gaussian Elimination with Partial Pivoting
    % and Back Substitution
    %
    function [X] = Gauss (A,b)
    %
    % Partial Pivot: If the second row is bigger than
    % the first row, swap them. The code is not scalable to
    % dimensions greater than 2.
    %
    if (A(2,1) > A(1,1))
        A([1,2],:) = A([2,1],:);
        b([1,2],:) = b([2,1],:);
    end
    %
    % Gaussian Elimination using the algorithm found
    % in the class notes
    %
    mult=A(2,1)/A(1,1);
    A(2,1)=A(2,1)-mult*A(1,1);
    A(2,2)=A(2,2)-mult*A(1,2);
    b(2)=b(2)-mult*b(1);
    %
    % Back Substitution. Again, the code is not very scalable.
    %
    x2=b(2)/A(2,2);
    x1=(b(1)-A(1,2)*x2)/A(1,1);
    X=[x1 x2];
    end
%
% Solutions: These were found using Wolfram Alpha.
% It is important to know the solutions in order to
% determine if we are getting close to them.
%
x_ans1=-1.92297337852049018l1;
y_ans1=0.549702997537298856l1;
x_ans2=1.768943856317070317 8;
y_ans2=-0.9331868158081167848 7;
%
% The format is set to long to improve accuracy. The norm
% and tolerance are initialized.
%
format long
LeNorme=1;
tol=10^(-8);
%
% 'm' is the number of steps, while b is the length
% of the square we will consider. Thus the step size will
% be 2b/m. L will contain the color information for each
% point in the square. x and y are all the points that
% will be iterated on.
%
m=256;
b=3;
L=zeros(m,m,3);
x=linspace(-b,b,m);
y=linspace(-b,b,m);
%
```

```
% The square region from -3 to 3 is scanned with a step size
% of 2b/m.
%
for p=1:m
    for q=1:m
        l=0;
        LeNorme=1;
        x0=x(p);
        y0=y(q);
        %
        % The algorithm for Newton's method found
        % in the class notes
        %
        while (1)
            l=l+1;
            J0=[2*x0, 2*y0; 2*(x0-1)/9, 2*(y0-1)/4];
            F=LeFonction(x0,y0);
            DeltaX=Gauss(J0,-F);
            x0=x0+DeltaX(1);
            y0=y0+DeltaX(2);
            LeNorme=(norm(DeltaX));
            % The max number of iterations is 200.
            if (l==200)
                L(q,p,3)=1;
                break
            end
            % If the tolerance is reached...
            if (LeNorme< tol)
                % ...and if we have converged to the first solution...
                if (abs(x0-x_ans1)<tol && abs(y0-y_ans1)<tol)
                % ...set the color to red, depending on the speed
                % of convergence. Notice that 'l' is the number of
                % iterations.
                    L(q,p,1)=(l/200)^(1/3);
                elseif (abs(x0-x_ans2)<tol && abs(y0-y_ans2)<tol)
                    L(q,p,2)=(l/200)^(1/3);
                else
                % Otherwise, mark it as blue.
                L(q,p,3)=1;
                end
                break
            end
        end
    end
end
%
% Initialize colormap matrix. Define r, which is the
% darkest color in the colormap. Set k to m/2 for convenience.
%
colors=zeros(m-1,3);
r=(1/200)^(1/3);
k=m/2;
%
% Between 1 and m/2, set the colors to red. The formula used below
% is found by knowing that at 1, color must be 1, (brightest), while
% at g=m/2 we must have the darkest shade, which has a value of r
%
```

```
for g=1:k;
    colors(g,1)=(g*(r-1))/(k-1)+1-((r-1)/(k-1));
end
%
% Similar to above, only now we need dark green at the beginning,
% and bright green at the end.
%
for h=k+1:m;
    colors(h,2)=(h*(r-1))/(k+1-m)+r-(k+1)*(r-1)/(k+1-m);
end
%
% Display the computed answers using imagesc
%
imagesc(x,y,L)
axis xy
colormap(colors)
colorbar
%
% The colormap limits are set like this because the
% max. number of iterations is set to 200 for both solutions.
% The negative sign indicates the red part of the colormap
%
caxis([-200, 200])
end
```

1.2.2 Plot of Basins of Attraction

From the algorithm above, two sets of solutions are:

$$\begin{cases} x_1 = -1.92297337 \\ y_1 = 0.549702997 \end{cases} and \begin{cases} x_2 = 1.768943856 \\ y_2 = -0.93318681 \end{cases}$$

Scanning the space $(x_0, y_0) \subset [-3;\ 3] \times [-3; 3]$,

- if (x_0, y_0) approaches (x_1, y_1), we colored it in red;
- if (x_0, y_0) approaches (x_2, y_2), we colored it in green;
- else we colored it in blue.

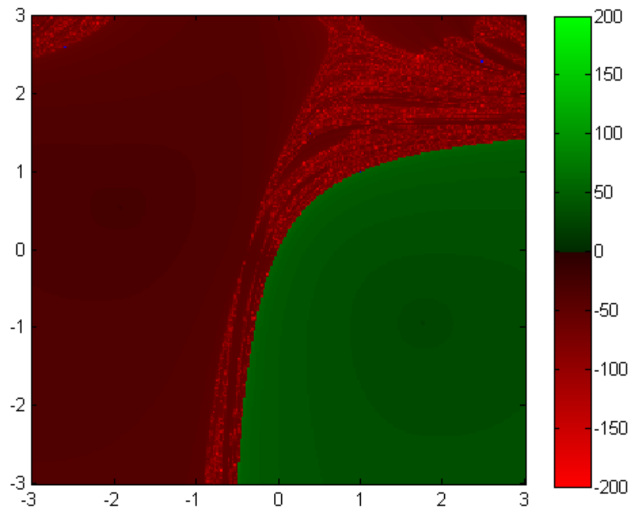

Figure 1: Basins of attraction using Newton's method to solve system 1

1.2.3 Comment

As can be seen from the above graph, there are almost no blue points present. This implies that Newton's method is relatively reliable when it comes to convergence to solutions. Each solution represents a basin of attraction. In general, the closer a point is to one of the solutions the faster it converges, as can be seen from the darker-colored regions around each basin. There are 2 regions in the graph with erratic convergence behaviour: one at the upper-left corner, and the other one spanning a large part of the center and the upper-right corner. Both of these erratic regions are red, meaning they both converge to solution 1, whereas there are no erratic regions for solution 2 (on this square, at least).

The boundary separating the two basins of attraction can be found by considering the algorithm for Newton's method; the Jacobian matrix must be computed at each iteration. If the starting point is chosen such that the Jacobian matrix is singular (i.e. det(J)=0), it will be impossible to solve the linear system of equations for finding $\Delta\mathbf{x}$. Thus:

$$\mathrm{J} = \begin{vmatrix} 2x & 2y \\ \frac{2}{9}(x-1) & \frac{1}{2}(y-1) \end{vmatrix}$$

$$\det(J) = \frac{9x}{5x+4} - y = 0$$

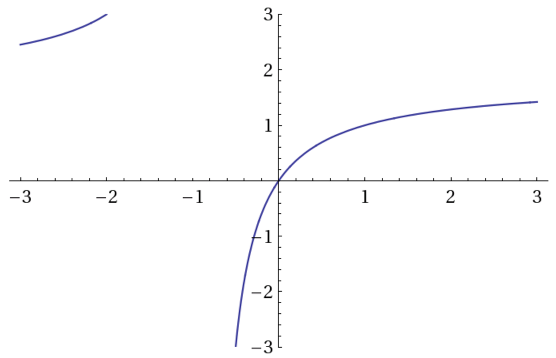

Figure 2: Plot of det (J)=0

However, the colored plot shows that there is no distinct blue line at the boundary between the two basins. This phenomenon can again be explained by considering the algorithm used. Not all points on the 3x3 square are scanned continuously; rather, they are limited by the step size we use. It is highly unlikely that we will iterate on a point that is exactly on the boundary between basins. If the point deviates even a little from the boundary, it will converge to one of the two basins instead of diverging. The inaccuracy of computer arithmetic makes it very difficult to be exactly on the boundary; therefore it does not appear in our plot.

1.3 Broyden's Method

Broyden's method is developed from adapting the algorithms of Newton's method. In the MATLAB algorithm in 1.3.1, Jacobian matrix is used as the initial guess. Plot of basins of attraction is shown in 1.3.2.

1.3.1 MATLAB Algorithm

```
function [] = broyden2()
    %
    % Define a function that carries out
    % Gaussian Elimination with Partial Pivoting
    % and Back Substitution
    %
    function [F] = LeFonction(u0,v0)
    fu0=u0^2+v0^2-4;
    fv0=((u0-1)^2)/9 + ((v0-1)^2)/4 -1;
    F=[fu0;fv0];
    end
    %
    % Define a function that carries out
    % Gaussian Elimination with Partial Pivoting
    % and Back Substitution
    %
    function [X] = Gauss (A,b)
    %
    % Partial Pivot: If the second row is bigger than
    % the first row, swap them. The code is not scalable to
    % dimensions greater than 2.
    %
    if (A(2,1) > A(1,1))
        A([1,2],:) = A([2,1],:);
        b([1,2],:) = b([2,1],:);
    end
    %
    % Gaussian Elimination using the algorithm found
    % in the class notes
    %
    mult=A(2,1)/A(1,1);
    A(2,1)=A(2,1)-mult*A(1,1);
    A(2,2)=A(2,2)-mult*A(1,2);
    b(2)=b(2)-mult*b(1);
    %
    % Back Substitution. Again, the code is not very scalable.
    %
    x2=b(2)/A(2,2);
    x1=(b(1)-A(1,2)*x2)/A(1,1);
    X=[x1 x2];
    end
%
% Solutions: These were found using Wolfram Alpha.
% It is important to know the solutions in order to
% determine if we are getting close to them.
%
```

```
x_ans1=-1.9229733785204901811;
y_ans1=0.54970299753729885611;
x_ans2=1.7689438563170703178;
y_ans2=-0.93318681580811678487;
%
% The format is set to long to improve accuracy. The norm
% and tolerance are initialized.
%
format long
LeNorme=1;
tol=10^(-8);
%
% 'm' is the number of steps, while b is the length
% of the square we will consider. Thus the step size will
% be 2b/m. L will contain the color information for each
% point in the square. x and y are all the points that
% will be iterated on.
%
m=256;
b=3;
L=zeros(m,m,3);
x=linspace(-b,b,m);
y=linspace(-b,b,m);
%
% The square region from -3 to 3 is scanned with a step size
% of 2b/m.
%
for p=1:m
    for q=1:m
        l=0;
        x0=x(p);
        y0=y(q);
        LeNorme=1;
        % Define the Jacobian and the iteration matrix A0
        A0=[1 0;0 1];
        J0=[2*x0, 2*y0; 2*(x0-1)/9, 2*(y0-1)/4];
        A0=J0;
        % The algorithm for Broyden's method found in the class notes.
        while (1)
            l=l+1;
            F0=LeFonction(x0,y0);
            DeltaX=Gauss(A0,-F0);
            x0=x0+DeltaX(1);
            y0=y0+DeltaX(2);
            F1=LeFonction(x0,y0);
            DeltaF=F1-F0;
            A0=A0+(((DeltaF-A0*DeltaX')*(DeltaX))/(norm(DeltaX)));
            LeNorme=norm(DeltaX);
            % Max. numebr of iterations is 200.
            if (l==200)
                L(q,p,3)=1;
                break
            end
            % If the tolerance is reached...
            if (LeNorme< tol)
                % ...and if we have converged to the first solution...
                if (abs(x0-x_ans1)<tol && abs(y0-y_ans1)<tol)
```

```
                    % ...set the color to red, depending on the speed
                    % of convergence. Notice that 'l' is the number of
                    % iterations.
                        L(q,p,1)=(l/200)^(1/3);
                    elseif (abs(x0-x_ans2)<tol && abs(y0-y_ans2)<tol)
                        L(q,p,2)=(l/200)^(1/3);
                    else
                    % Otherwise, mark it as blue.
                    L(q,p,3)=1;
                    end
                    break
                end
            end
        end
    end
    %
    % Initialize colormap matrix. Define r, which is the
    % darkest color in the colormap. Set k to m/2 for convenience.
    %
    colors=zeros(m-1,3);
    r=(1/200)^(1/3);
    k=m/2;
    %
    % Between 1 and m/2, set the colors to red. The formula used
    % below is found by knowing that at 1, color must be 1, (brightest),
    % while at g=m/2 we must have the darkest shade, which has a value of
    r
    %
    for g=1:k;
        colors(g,1)=(g*(r-1))/(k-1)+1-((r-1)/(k-1));
    end
    %
    % Similar to above, only now we need dark green at the beginning,
    % and bright green at the end.
    %
    for h=k+1:m;
        colors(h,2)=(h*(r-1))/(k+1-m)+r-(k+1)*(r-1)/(k+1-m);
    end
    %
    % Display the computed answers using imagesc
    %
    imagesc(x,y,L)
    axis xy
    colormap(colors)
    colorbar
    caxis([-200, 200])
    end
```

1.3.2 Plot of Basins of Attraction

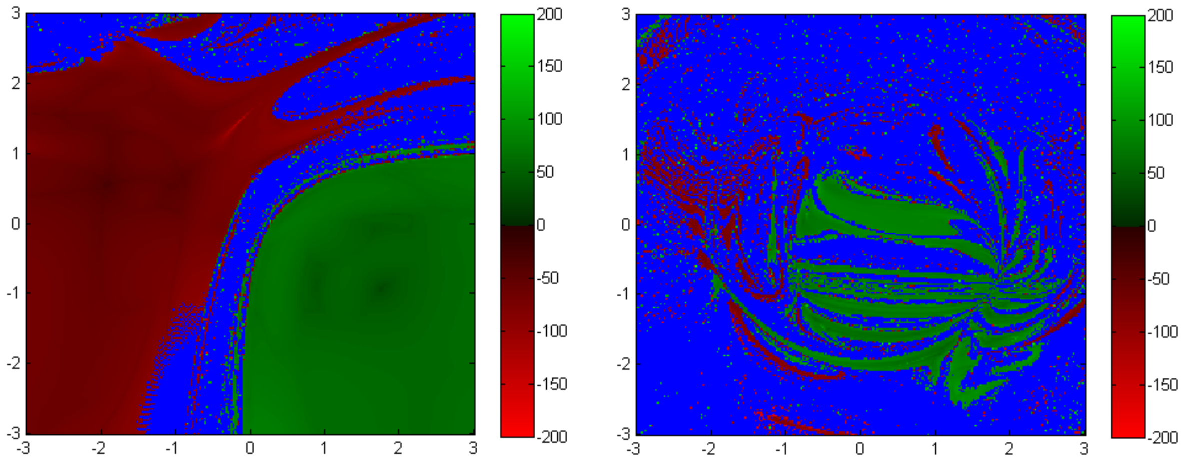

Figure 3 &4 : Basins of attraction using Broyden's method
with initial guesses of Jacobian matrix (left) and identity matrix (right)

1.3.3 Comments

Same plotting criteria as 1.2.2 in Newton's method are used here. In Figure 3, the darker region of green and red converges faster than the lighter region. Comparing Figure 3 and Figure 4 which are computed from initial guess of Jacobian matrix and identity matrix respectively, it is clear that using the identity matrix as the initial guess leads to a significant amount of non-converged solutions.

This discrepancy can be explained by considering the algorithm used in Broyden's method. It involves solving the linear system of equations $\mathbf{A_0\Delta x=-f(x_0)}$. If the Jacobian matrix is used as an initial guess, we have a good indication of where the solution is headed due to the information contained in the Jacobian (i.e. the partial the derivatives of the functions). If, on the other hand, we use the identity matrix as our initial guess, we only have the initial point x_0 available to indicate the direction of the solution. This causes the lack of convergence observed as blue points in Figure 4.

2 Audio time-stretching

2.1 Interpolation or Approximation?

In order to replicate and reduce the speed of a digital audio signal by a factor of 2, we chose to interpolate the original audio signal file by natural cubic spline.

2.1.1 Explanation of choice

Both interpolation and approximation are useful methods for analyzing data points, but based on the purpose of this assignment interpolation would be more suitable. We are explicitly required to stretch the original sample, and the way this will be accomplished is by adding inaccessible intermediate points in-between existing ones. A curve must therefore be fitted on the existing data before new data is added. By having more points in the sample, the sound will last for a longer time if it is played at the same frequency as before.

By contrast, approximation is more suitable where experimental error is of concern in order to smooth out undesired variations. Since this is not the case for this project, interpolation will be used instead. Specifically, the interpolation will be carried out via piecewise cubic splines. It is fast, efficient, and stable, and provides a good compromise between simplicity and functional richness.

2.1.2 MATLAB Algorithm

```
clc;
factor=2;
%Read sample wav file
[original,Fs,nbits]=wavread('testing6.wav');
original=original(:,1);
%store into original array
%count number of elements in the original sample.wav
n=length(original);
Wave_sample_rate=Fs;
Wave_bits=nbits;
%duration of the original sample in seconds
duration=n/Fs;
%x is an evenly spaced from 0 to duration with n elements
x=linspace(0,duration,n)';
%
%Interpolation with natural cubic spline
%Use the pseudo code as shown in the txt book
a=zeros(n,1);
l=zeros(n,1);
u=zeros(n,1);
z=zeros(n,1);
c=zeros(n,1);
b=zeros(n,1);
d=zeros(n,1);
for i=1:n;
```

```
    a(i)=original(i);
end
%h is the constant step
h=x(2)-x(1);
alpha=zeros(n-1,1);
for i=2:1:n-1
     alpha(i)=(3/h)*(a(i+1)-a(i))-(3/h)*(a(i)-a(i-1));
end
l(1)=1;
u(1)=0;
z(1)=0;
for i=2:1:n-1;
    l(i)=2*(x(i+1)-x(i-1))-h*u(i-1);
    u(i)=h/l(i);
    z(i)=(alpha(i)-h*z(i-1))/l(i);
end
l(n)=1;
z(n)=0;
c(n)=0;
for j=n-1:-1:1;
    c(j)=z(j)-u(j)*c(j+1);
    b(j)=(a(j+1)-a(j))/h-h*(c(j+1)+2*c(j))/3;
    d(j)=(c(j+1)-c(j))/(3*h);
end
s=zeros(n-1,1);
for j=1:n-1;
    k=x(j)+h/factor;
    while k<x(j+1);
        s(j)=a(j)+b(j)*(k-x(j))+c(j)*(k-x(j))^2+d(j)*(k-x(j))^3;
        k=k+h/factor;
    end
end
%
%Time-stretching
interpo_n=factor*n;
interpo_wave=zeros(interpo_n,1);
%x_interpo is an evenly spaced from 0 to new duration with interpo_n
elements
x_interpo=linspace(0,factor*duration,interpo_n)';
i=1;
j=1;
%interpolation wave
while i<=interpo_n;
    interpo_wave(i)=original(j);
    interpo_wave(i+1)=s(j);
    i=i+factor;
    j=j+1;
    if j>=n-1;
        break
    end
end
%
%Plot
subplot(2,1,1);
plot(x,original);
xlabel('Time(s)');
ylabel('Amplitude');
```

```
title('Original Signal before Time-stretching');
axis([0 nbits -1 1]);
subplot(2,1,2);
plot(x_interpo,interpo_wave);
xlabel('Time(s)');
ylabel('Amplitude');
title('Singal after Natural Cubic Spline Interpolation');
axis([0 nbits -1 1]);
%Write the interpo_wave to Interpolation.wav at original rate and
numbers of bits
wavwrite(interpo_wave,Fs,nbits,'Interpolation.wav')
%play the interpolated file after time-stretching
soundsc(interpo_wave,Fs)
```

The above algorithm produces the wave plot in Figure 5:

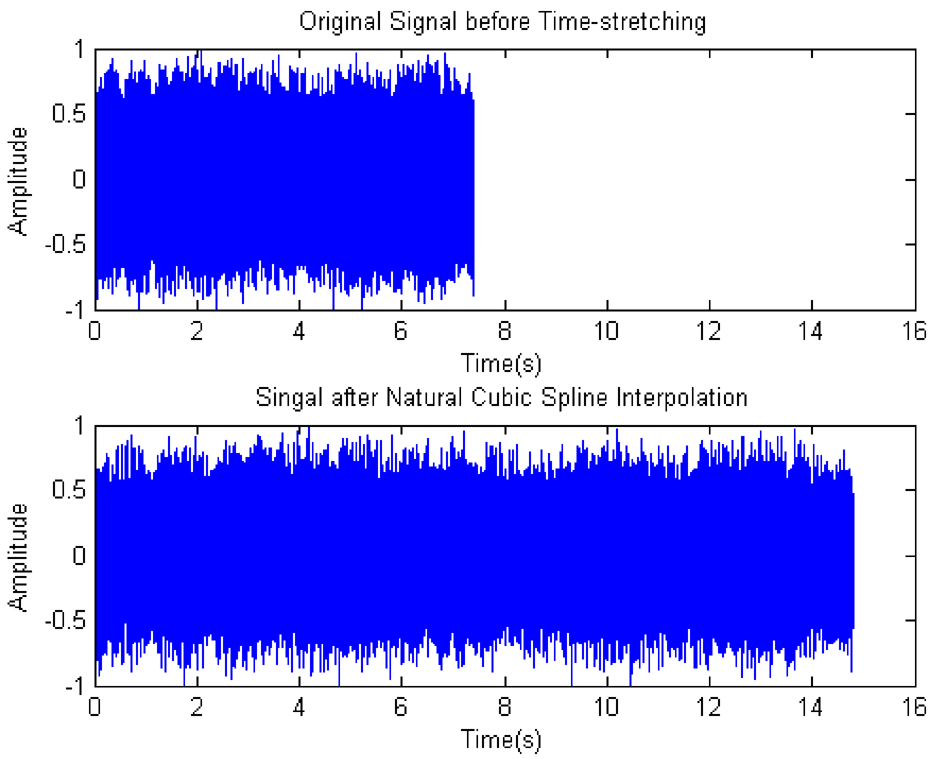

Figure 5: Comparison between the original and time-stretched signal after natural cubic spline interpolation

2.1.3 Main drawback

The biggest drawback of spline interpolation is the lack of precision. By comparing the audio files (testing6.wav and interpolation.wav), it is very obvious to hear the difference. That means the spline interpolation does not replicate the sound wave well enough. They could oscillate around an outlier which causes instability of the interpolation. Since outliers can be fairly common in audio samples, this can pose a significant problem depending on the sound file used.

2.2 Granular synthesis

In the last section we introduced natural cubic spline interpolation for a sample audio signal. Though it is easy and intuitive, the main disadvantage of that method causes the signal after time-stretching is not as precise as the original one. In this section, a more sophisticated method called granular synthesis will be performed. The sample signal is split into small pieces called grains. And then these grains are layered on top of each other and recombined to a different time-scale of signal.

2.2.1 MATLAB Algorithm

```
function [] = granular (filename,N2,N3,M)
% Syntax: granular('testing.wav',10000,20000,15000)
%
%
% This is the window function. It takes an empty array as
% input and fills it with elements from the original sample.
% The information for this window function is found in the
% project description.
%
function [C] = window(C)
% The first interval
lower=N1;
higher=N1+N2;
for i=lower:higher;
    % This 'if' statement is needed in order to avoid
    % going beyond the limits of the array 'Y'
    if i>=numel(Y)
        break
    end
    C(i-M*(k-1)) = Y(i)*((i-N1)/N2);
end
% The second interval
lower=N1+N2;
higher=N1+N2+N3;
for i=lower:higher;
    if i>=numel(Y)
        break
    end
    C(i-M*(k-1)) = Y(i);
end
% The third interval
lower=N1+N2+N3;
higher=N1+2*N2+N3;
for i=lower:higher;
    if i>=numel(Y)
        break
    end
    C(i-M*(k-1)) = Y(i)*((N1+2*N2+N3-i)/N2);
end
end
%
```

```
% Reads the file and makes sure it is monophonic
% by only retaining the first column of samples
%
Y=wavread(filename);
Y=Y(:,1);
% The windows have the same length
wlength=2*N2+N3;
%
wavsize=size(Y);
wavsize=wavsize(1,1);
% The number of windows is given by the following formula.
% This formula was determined by inspection.
N_windows=(wavsize-(wlength))/M +1;
N_windows=N_windows(:,1);
N_windows=round(N_windows);
%
% Initializes the matrix that contains all the windows.
% Each column represents the data points of 1 window.
%
W=zeros(wlength,N_windows);
%
% In order to start reading windows from the beginning of the audio
signal,
% we need to set N1=1. This value will then be updated by M
% during the following loop in order to get subsequent windows.
%
N1=1;
%
for k=1:N_windows;
    % Initializes a temporary array that contains the data points
    % for the current window.
    A=zeros(wlength,1);
    % This empty array is filled up by calling on the window function.
    A=window(A);
    % The elements contained in A are now copied into the appropriate
    % column of W.
    for l=1:wlength;
        W(l,k)=A(l);
    end
    % The next window is found by shifting the index of N1 by M
    N1=N1+M;
end
%
% The number 'z' is equal to the total length of the stretched audio
% sample. This formula was found by inspection.
%
z=N2+(N2+N3)*N_windows;
%
% Initializes the stretched vector. It is 1-dimensional just like
% the original sample 'Y'.
%
B=zeros(z,1);
% These values represent the starting and ending points in 'B'.
% Initially, we will put the first window at the beginning of 'B'
smallest=1;
biggest=wlength;
% The following loops will recombine all the windows into
% the stretched signal.
for m=1:N_windows;
    % The following formula was again foudn by inspection.
    for j=smallest:biggest;
        B(j)=B(j)+W(j-(m-1)*(N2+N3),m);
    end
    % The next window is not put at the beginning of 'B', but
    % rather where the last window left off.
    smallest=smallest+N2+N3;
    biggest=biggest+N2+N3;
end
%
% Define time domains
time1=(1/44100)*(wavsize);
time2=(1/44100)*z;
t1=linspace(0,time1,wavsize);
t2=linspace(0,time2,z);
% Plots original sample
subplot(2,1,1)
plot(t1,Y)
xlabel('Time(s)');
ylabel('Amplitude');
title('Original Signal before Time-stretching');
axis([0 time1 -1 1])
% Plots stretched sample
subplot(2,1,2)
plot(t2,B)
xlabel('Time(s)');
ylabel('Amplitude');
title('Signal after Granular Synthesis');
axis([0 time1 -1 1])
% Outputs the stretched audio file
wavwrite(B,44100,'granular.wav')
% Outputs the stretching factor alpha
alpha=size(B)/wavsize;
% Outputs the time-stretching factor
alpha=alpha(1,1)
end
```

2.2.2 Plots

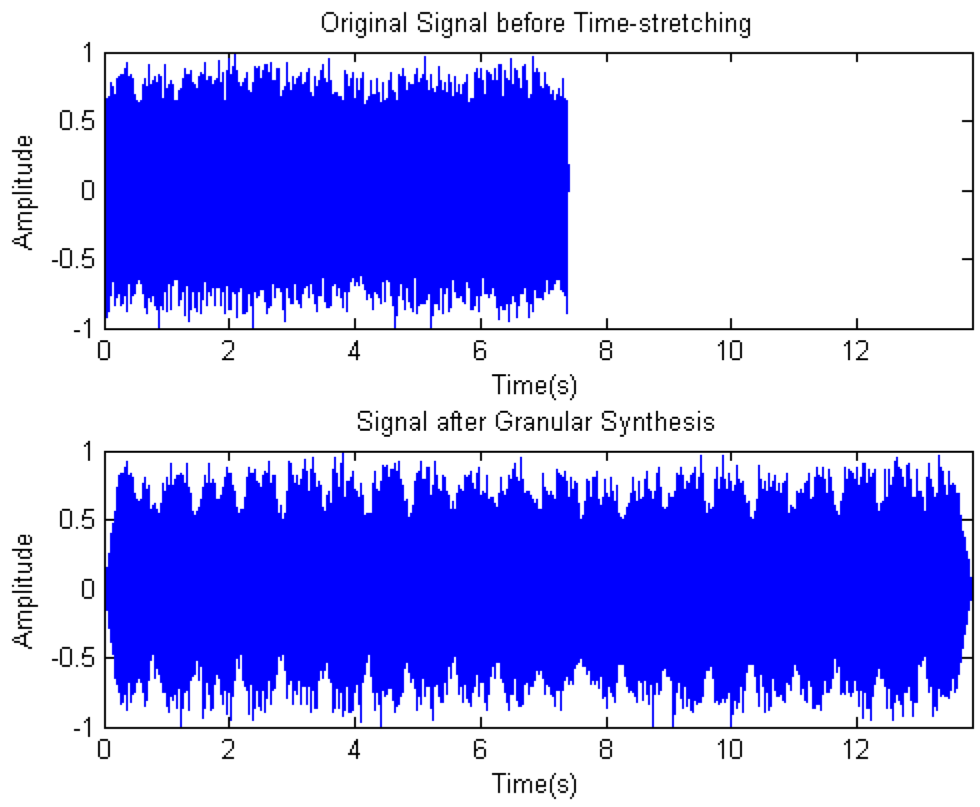

Figure 6: Comparison between the original and time-stretched signal after Granular synthesis when M=15000

The stretched sample was increased by a factor of α=1.8707. The values used for N_2, N_3, M were 10000, 20000, 15000, respectively. The value of N_1 is not specifiable by the user, but is automatically updated by the MATLAB code for each window. Notice that there is a loss in amplitude for the very first and very last grain due to the formula used in the definition of grains.

2.2.3 Advantage over interpolation

The main advantage of granular synthesis over interpolation is in terms of computational costs. Granular synthesis is much faster than interpolation because it essentially does not involve any computation except for adding the windows together during the merging process, whereas interpolation requires the computation of many spline equations. The precision problems inherent with cubic spline interpolation are also absent from granular synthesis, namely the presence of outliers in the signal. Also by listening to the output audio files from two different methods, the one created after granular synthesis sounds much alike as the original sample.

2.2.4 Duration reduction

To reduce the duration of the sample, it is necessary to pick an M such that

$$N_2 + N_3 < M < 2N_2 + N_3$$

To see why, consider Figure 3 in the project description document. If M is less than N_2+N_3, the grain (i.e. window) will be shifted to the right in order to be merged, thus lengthening the signal. If it is chosen bigger than $2N_2$+N_3, part of the original signal is completely lost and is not present in the stretched audio file. If, however, M is in between these two values, the original signal is kept in its entirety, and the duration of the sample is reduced.

The following example has a value of α=0.8587, with N_2=10000, N_3=20000, M=35000.

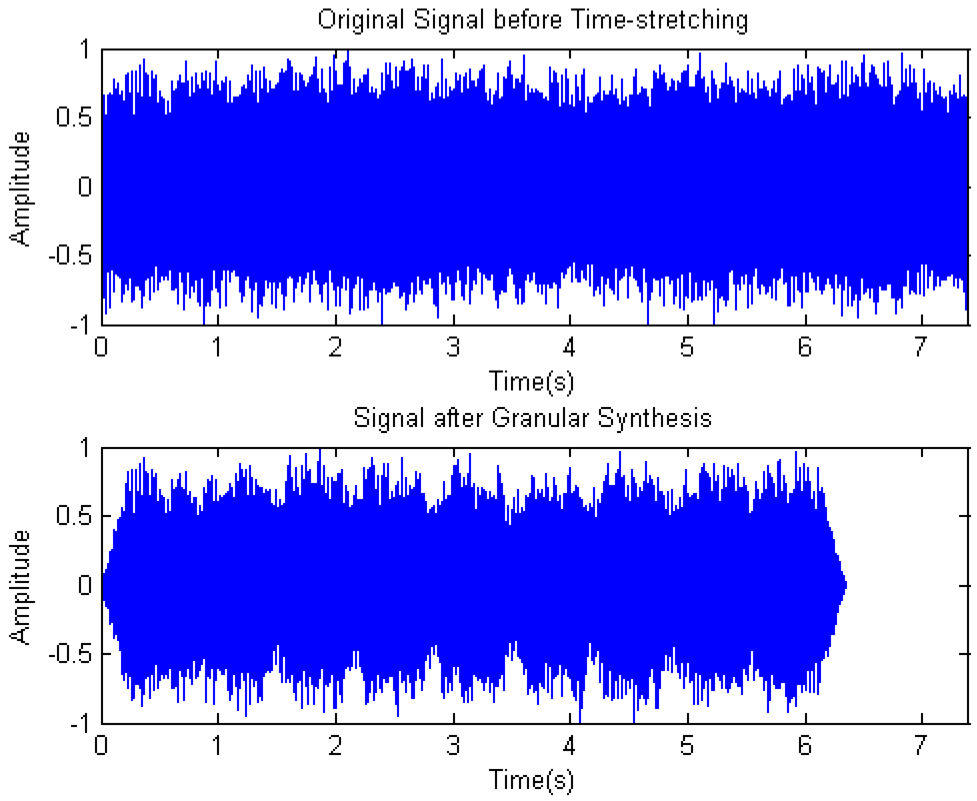

Figure 7: Comparison between the original and time-stretched signal after Granular synthesis when M=35000