Finding Structure in Turbulent Flows via DMD: An AMATH 582 Final Project

Jesse Dumas

September 14, 2018

This is a discussion on the dynamic mode decomposition (DMD) algorithm and its use in discovering dynamic structure in turbulent flows. Direct numerical simulation (DNS) data, publicly available from the Johns Hopkins Turbulence Databases (JHTDB), is explored using MATLAB. The method successfully tracked vortex cores through low pressure regions and time dependent structure in the velocity field. However, the known shortcomings of DMD relating to invariance became apparent while tracking vortices.

Introduction and Overview

Fluid dynamicists have traditionally used proper orthogonal decomposition (POD)–or variants of the method–to analyze flow data for coherent structures. Though the method is effective, it falls short in two areas: (a) the energy from singular value decomposition (SVD) is not always the best measure for ranking coherent structures, and (b) important phase information is discarded in the process[1].

Dynamic mode decomposition was developed by Peter Schmidt to circumvent these POD pitfalls, particularly for experimental data where velocimetry measurements might be the only input available for analysis. For a linearized flow, the results from DMD are equivalent to POD, and for a nonlinear flow (i.e. a turbulent flow) the method produces structures of a linear tangent approximation of the underlying flow, similar to Koopman analysis of nonlinear dynamical systems[2].

This project was an attempt at applying DMD on direct numerical simulation (DNS) data to find coherent structures in a turbulent flow. The data were from the Johns Hopkins Turbulence Databases, a public collection of DNS and large eddy simulation data sets with a MATLAB API. I explored the forced isotropic turbulence data set (extended)[3,4], a $2\pi \times 2\pi \times 2\pi$ domain divided into 1024^3 grid points with a Taylor-scale Reynolds number of $Re_\lambda = u'\lambda/\nu = 418$, where u' is the RMS velocity, λ is the Taylor micro-scale (the size of the smallest eddy), and ν is the kinematic viscosity of the fluid.

[1] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010

[2] Igor Mezić. Spectral properties of dynamical systems, model reduction and decompositions. *Nonlinear Dynamics*, 41(1):309–325, 2005

[3] M. Wan Y. Yang R. Burns C. Meneveau R. Burns S. Chen A. Szalay G. Eyink Y. Li, E. Perlman. A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, 9(31), 2008

[4] Eric Perlman, Randal Burns, Yi Li, and Charles Meneveau. Data exploration of turbulence simulations using a database cluster. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 23:1–23:11, New York, NY, USA, 2007. ACM

Theoretical Background

Navier-Stokes Equations

The fundamental equations for an incompressible, viscous flow are the Navier-Stokes equations, simplified as follows:

$$\nabla \cdot \vec{u} = 0\,, \tag{1}$$

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\vec{u} + \vec{F}\,, \tag{2}$$

Despite looking fairly simple, the equations are highly nonlinear. Memory requirements scale as $Re_\lambda^{9/4}$, restricting the size of direct numerical simulations like the one used for this project.

Dynamic Mode Decomposition[5]

[5] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. Dynamic mode decomposition: Data-driven modeling of complex systems, 2016

In simple terms, DMD takes data from a dynamical system

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t, \mu)\,, \tag{3}$$

where $\mathbf{f}$ represents the dynamics of the system based on the state, $\mathbf{x}(t)$, evolving in time, t, with parameters, μ, and constructs an approximate locally linear system

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x}\,. \tag{4}$$

With initial condition $\mathbf{x}(0)$, the solution of Equation 4 is

$$\mathbf{x}(t) = \sum_{k=1}^{n} \phi_k exp(\omega_k t) b_k = \Phi exp(\Omega t) b\,, \tag{5}$$

where ϕ_k and ω_k are eigenvectors and eigenvalues of $\mathbf{A}$. To speed the computation, $\tilde{\mathbf{A}}$, a projection of $\mathbf{A}$ onto the dominant POD modes, is used. The outputs of the decomposition are the eigenvalues (DMD eigenvalues) and eigenvectors (DMD modes) of $\mathbf{A}$. Figure 1 illustrates the DMD procedure for an example fluid dynamics problem.

Algorithm Implementation and Development

Obtaining Data

Flow field data were obtained using the Turbmat MATLAB interface from JHTDB [6]. The package includes functions for retrieving fields and quantities from the databases at varying times and points. I opted to analyze 2D slices of the full data to simplify my process.

[6] `http://turbulence.pha.jhu.edu/help/matlab/`

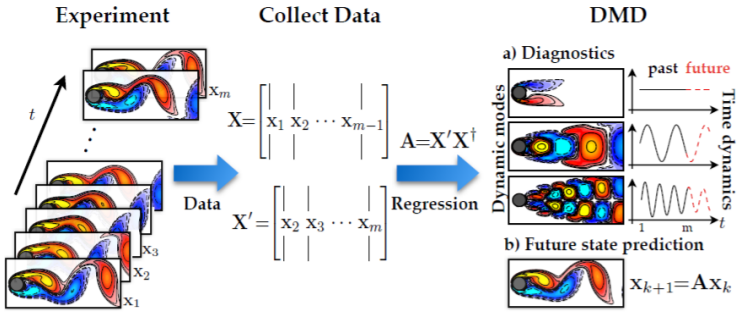


Figure 1: A schematic example of DMD on a fluid problem from Dynamic Mode Decomposition: Data Driven Modeling of Complex Systems, 2016.

The Algorithm

1. Get time step.
2. Import velocity field for a small 64×64 subdomain of the 1024^3 grid for 100 timesteps.
3. Calculate velocity magnitude from velocity field, $|\mathbf{v}| = \sqrt{u_x^2 + u_y^2 + u_z^2}$.
4. Reshape 2D velocity magnitude field into column vectors.
5. Build matrix $\mathbf{X}$ with each column being a snapshot of the flow at a different time t.
6. Take the SVD of $\mathbf{X}$. $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^*$.
7. Compute the matrix $\mathbf{A}$ from the pseudo-inverse of $\mathbf{X}$.
8. Compute the matrix $\tilde{\mathbf{A}}$, the $r \times r$ projection of $\mathbf{A}$ onto POD modes.
9. Compute the eigendecomposition, $\mathbf{W}$, of $\tilde{\mathbf{A}}$.
10. Reconstruct the eigendecomposition of $\mathbf{A}$ from $\mathbf{W}$ and Λ.
11. Subtract the static objects from the original data.
12. Reshape result to visualize the DMD modes.
13. Repeat process for pressure field.

The full code for the project is presented in Appendix B.

Computational Results

Tracking coherent structures in the DNS data proved challenging. As the following sections show, extracting structures from the pressure field provided better results.

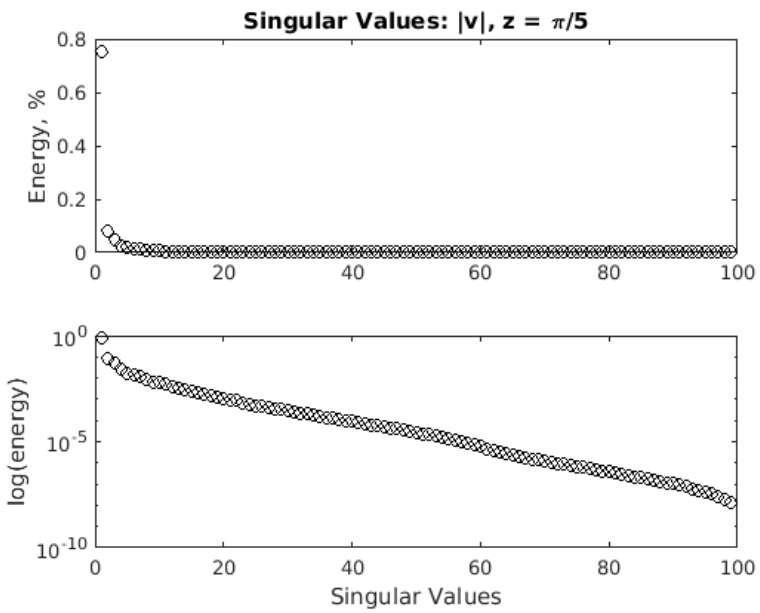


Figure 2: Singular values for the velocity magnitude. There is one dominant mode, indicating that the data can be projected to a very low rank space.

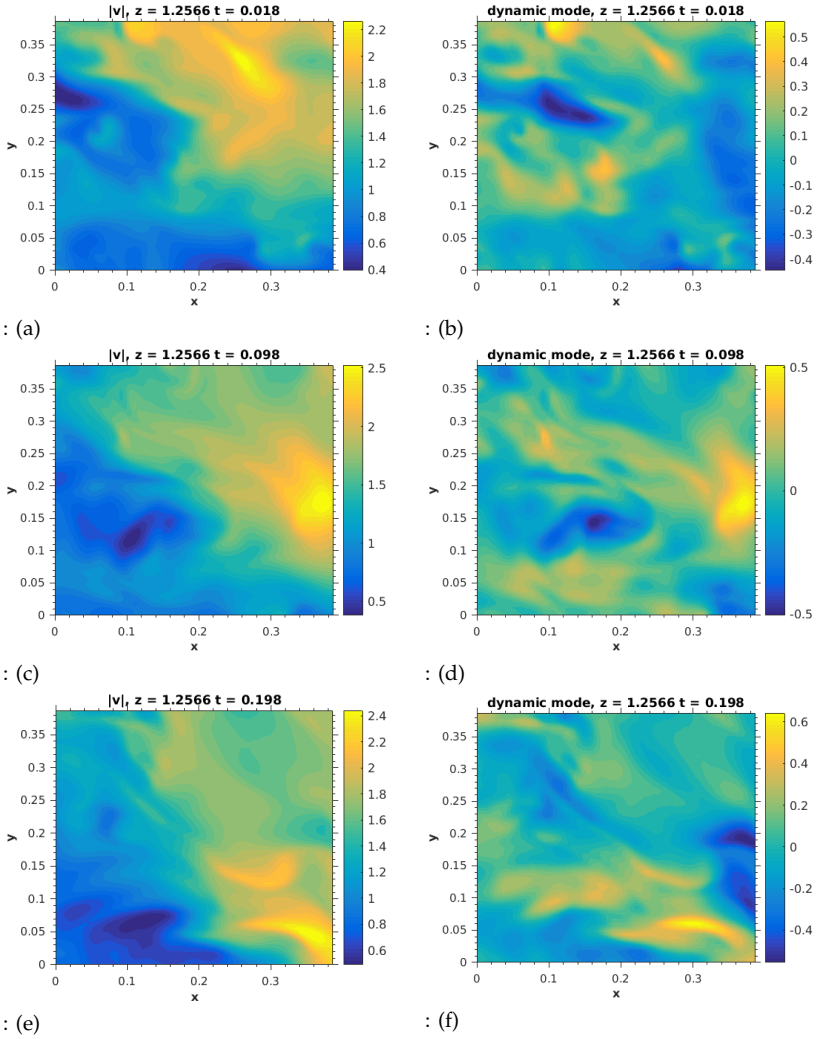
Velocity Field

As shown in Figure 2, there is one dominant mode in the data. However, it only accounts for about 90% of the energy, while the remaining 10% is in another three or so modes. Looking at the modes plotted on a semi-log plot in the bottom half of Figure 2 makes it clear that the modes do not drop off as quickly as it first appeared. There are two explanations for this. The first is that turbulent flows contain coherent structures that span several orders of magnitude in characteristic length, and DMD might just be extracting more than one of these scales as important. The second explanation is that invariance in data is one of DMD's weaknesses[7], and turbulent flows are highly invariant. Figure 4 illustrates the attempt at extracting coherent structures from the velocity magnitude.

[7] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. Dynamic mode decomposition: Data-driven modeling of complex systems, 2016

Pressure Field

Extracting coherent structures from the pressure field seems to have proven more fruitful. Figure 7 shows DMD apparently singling out low-pressure vortex cores in the flow while discarding most of the flow outside those cores. Figure 5 shows the singular values for the pressure data dropping off rapidly, leaving one dominant mode.



: (a) : (b)

: (c) : (d)

: (e) : (f)

Figure 4: Left column: velocity magnitude evolving in time. Right column: DMD modes, an attempt at tracking flow structures.

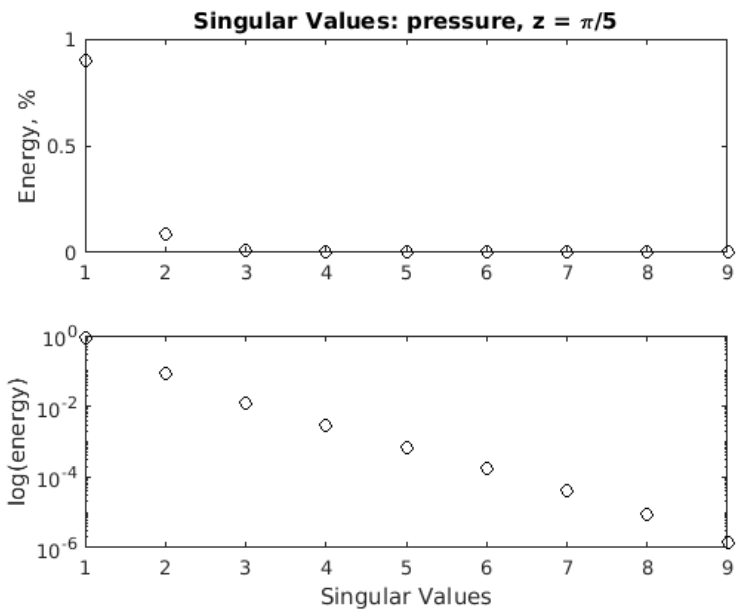


Figure 5: Singular values for the velocity magnitude. There is one dominant mode, indicating that the data can be projected to a very low rank space.

Summary and Conclusions

Building a DMD algorithm to extract coherent structures was not entirely successful, but it was still a valuable learning experience. As is noted in the literature[8], DMD and other SVD-based models struggle with handling invariance in data. Any data from a turbulent flow is guaranteed to be invariant–not only translationally, but also rotationally–as vortex structures rotate, translate, grow, and merge. This has sparked my interest in finding and reading more on methods that overcome this weakness.

[8] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. Dynamic mode decomposition: Data-driven modeling of complex systems, 2016

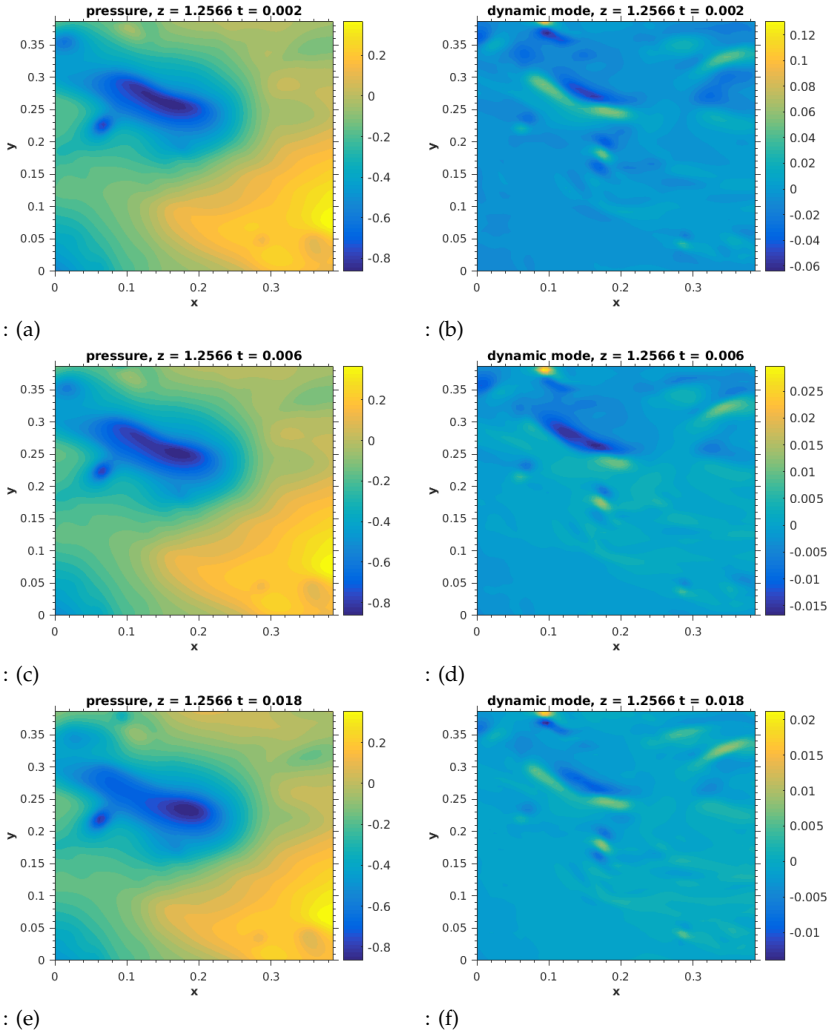


Figure 7: Left column: pressure field evolving in time. Right column: DMD modes, an attempt at tracking vortex cores.

References

[1] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

[2] Igor Mezić. Spectral properties of dynamical systems, model reduction and decompositions. *Nonlinear Dynamics*, 41(1):309–325, 2005.

[3] M. Wan Y. Yang R. Burns C. Meneveau R. Burns S. Chen A. Szalay G. Eyink Y. Li, E. Perlman. A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, 9(31), 2008.

[4] Eric Perlman, Randal Burns, Yi Li, and Charles Meneveau. Data exploration of turbulence simulations using a database cluster. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 23:1–23:11, New York, NY, USA, 2007. ACM.

[5] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. Dynamic mode decomposition: Data-driven modeling of complex systems, 2016.

Appendix A: MATLAB Functions

```
double() -- converts data type to double precision floating point.
load() -- loads saved data.
size() -- returns dimensions of a matrix.
[U,S,V] = svd(A); -- performs SVD.
my_dmd -- returns the background stream from a video via DMD.
vidfunc -- processes a video and performs DMD by calling my_dmd.
getVelocity -- obtains velocity field from a particular simulation.
getPressure -- obtains pressure field from a particular simulation.
```

Appendix B: MATLAB Code

The following script calls my_dmd.m to complete the homework and build plots.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Final Project for AMATH 582
% Jesse Dumas
% Coherent flow structures via DMD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all; close all; clc
```

```
load z_pi_on_5.mat;
load press_test.mat

% DMD for velocity
X = vel_data;
x1 = X(:,1:end-1);
x2 = X(:,2:end);
r=2;
dt=0.002;
[Phi,omega,lambda,b,Xdmd,S,U] = my_dmd(x1,x2,r,dt);

sig = diag(S)/sum(diag(S));

figure(1);
subplot(2,1,1), plot(sig,'ko','Linewidth',[1.1])
title('Singular Values: |v|, z = \pi/5')
ylabel('Energy, %')
subplot(2,1,2), semilogy(sig,'ko','Linewidth',[1.1])
ylabel('log(energy)')
xlabel('Singular Values')

dynamic_modes = X(:,1:99) - Xdmd;

spacing = 2.0*pi/1023;
nx = 64;
ny = nx;
xoff = 2*pi*rand;
yoff = 2*pi*rand;
zoff = 2*pi*0.2;
npoints = nx*ny;
clear points;

% Create surface
x = linspace(0, (nx-1)*spacing, nx);% + xoff;
y = linspace(0, (ny-1)*spacing, ny);% + yoff;
[X Y] = meshgrid(x, y);
points(1:2,:) = [X(:)'; Y(:)'];
points(3,:) = zoff;

time = 0.002*99; %

Z = reshape(real(dynamic_modes(:,99)), 64, 64);

figure(3)
```

```
contourf(X, Y, Z, 30, 'LineStyle', 'none');
set(gca, 'FontSize', 11)
title(['dynamic mode, z = ', num2str(zoff), ' t = ' num2str(time)], 'FontSize', 13, 'FontWeight', 'bold');
xlabel('x', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('y', 'FontSize', 12, 'FontWeight', 'bold');
colorbar('FontSize', 12);
axis([0 max(x) 0 max(y)]);
set(gca, 'TickDir', 'out', 'TickLength', [.02 .02],'XMinorTick', 'on', 'YMinorTick', 'on');

Z2 = reshape(vel_data(:,99), 64, 64);

figure(4)
contourf(X, Y, Z2, 30, 'LineStyle', 'none');
set(gca, 'FontSize', 11)
title(['|v|, z = ', num2str(zoff), ' t = ' num2str(time)], 'FontSize', 13, 'FontWeight', 'bold');
xlabel('x', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('y', 'FontSize', 12, 'FontWeight', 'bold');
colorbar('FontSize', 12);
axis([0 max(x) 0 max(y)]);
set(gca, 'TickDir', 'out', 'TickLength', [.02 .02],'XMinorTick', 'on', 'YMinorTick', 'on');

% DMD for pressure
X = pressure_data;
x1 = X(:,1:end-1);
x2 = X(:,2:end);
r=2;
dt=0.002;
[Phi,omega,lambda,b,Xdmd,S,U] = my_dmd(x1,x2,r,dt);

sig = diag(S)/sum(diag(S));

figure(1);
subplot(2,1,1), plot(sig,'ko','Linewidth',[1.1])
title('Singular Values: pressure, z = \pi/5')
ylabel('Energy, %')
subplot(2,1,2), semilogy(sig,'ko','Linewidth',[1.1])
ylabel('log(energy)')
xlabel('Singular Values')

dynamic_modes = X(:,1:9) - Xdmd;

spacing = 2.0*pi/1023;
nx = 64;
ny = nx;
```

```
xoff = 2*pi*rand;
yoff = 2*pi*rand;
zoff = 2*pi*0.2;
npoints = nx*ny;
clear points;

% Create surface
x = linspace(0, (nx-1)*spacing, nx);% + xoff;
y = linspace(0, (ny-1)*spacing, ny);% + yoff;
[X Y] = meshgrid(x, y);
points(1:2,:) = [X(:)'; Y(:)'];
points(3,:) = zoff;

time = 0.002*9; %

Z = reshape(real(dynamic_modes(:,9)), 64, 64);

figure(3)
contourf(X, Y, Z, 30, 'LineStyle', 'none');
set(gca, 'FontSize', 11)
title(['dynamic mode, z = ', num2str(zoff), ' t = ' num2str(time)], 'FontSize', 13, 'FontWeight', 'bold');
xlabel('x', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('y', 'FontSize', 12, 'FontWeight', 'bold');
colorbar('FontSize', 12);
axis([0 max(x) 0 max(y)]);
set(gca, 'TickDir', 'out', 'TickLength', [.02 .02],'XMinorTick', 'on', 'YMinorTick', 'on');

Z2 = reshape(vel_data(:,9), 64, 64);

figure(4)
contourf(X, Y, Z2, 30, 'LineStyle', 'none');
set(gca, 'FontSize', 11)
title(['pressure, z = ', num2str(zoff), ' t = ' num2str(time)], 'FontSize', 13, 'FontWeight', 'bold');
xlabel('x', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('y', 'FontSize', 12, 'FontWeight', 'bold');
colorbar('FontSize', 12);
axis([0 max(x) 0 max(y)]);
set(gca, 'TickDir', 'out', 'TickLength', [.02 .02],'XMinorTick', 'on', 'YMinorTick', 'on');
```

This is how I obtained the data.

```
%
% Turbmat - a Matlab library for the JHU Turbulence Database Cluster
%
```

```
% Sample code, part of Turbmat
%

%
% Written by:
%
% Jason Graham
% The Johns Hopkins University
% Department of Mechanical Engineering
% jgraha8@gmail.com
%

%
% Modified by:
%
% Edo Frederix
% The Johns Hopkins University / Eindhoven University of Technology
% Department of Mechanical Engineering
% edofrederix@jhu.edu, edofrederix@gmail.com
%

%
% This file is part of Turbmat.
%
% Turbmat 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 (at your option)
% any later version.
%
% Turbmat 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.
%
% You should have received a copy of the GNU General Public License along
% with Turbmat.  If not, see <http://www.gnu.org/licenses/>.
%
% Modified some more by Jesse Dumas for AMATH 582 final project

clear all;
close all;

authkey = 'edu.jhu.pha.turbulence.testing-201406';
dataset = 'isotropic1024coarse';
```

```
% ---- Temporal Interpolation Options ----
NoTInt   = 'None' ; % No temporal interpolation
PCHIPInt = 'PCHIP'; % Piecewise cubic Hermit interpolation in time

% ---- Spatial Interpolation Flags for getVelocity & getVelocityAndPressure ----
NoSInt = 'None'; % No spatial interpolation
Lag4   = 'Lag4'; % 4th order Lagrangian interpolation in space
Lag6   = 'Lag6'; % 6th order Lagrangian interpolation in space
Lag8   = 'Lag8'; % 8th order Lagrangian interpolation in space

% ---- Spatial Differentiation & Interpolation Flags for getVelocityGradient & getPressureGradient ----
FD4NoInt = 'None_Fd4' ; % 4th order finite differential scheme for grid values, no spatial interpolation
FD6NoInt = 'None_Fd6' ; % 6th order finite differential scheme for grid values, no spatial interpolation
FD8NoInt = 'None_Fd8' ; % 8th order finite differential scheme for grid values, no spatial interpolation
FD4Lag4  = 'Fd4Lag4'  ; % 4th order finite differential scheme for grid values, 4th order Lagrangian inter

% ---- Spline interpolation and differentiation Flags for getVelocity,
% getPressure, getVelocityGradient, getPressureGradient,
% getVelocityHessian, getPressureHessian
M1Q4   = 'M1Q4'; % Splines with smoothness 1 (3rd order) over 4 data points. Not applicable for Hessian.
M2Q8   = 'M2Q8'; % Splines with smoothness 2 (5th order) over 8 data points.
M2Q14   = 'M2Q14'; % Splines with smoothness 2 (5th order) over 14 data points.

%  Set time step to sample
time = 0.364;

% getPosition integration settings
startTime=0.364;
endTime=0.376;
lagDt=0.0004;

npoints = 10;

% for box filtering
field = 'velocity';
scalar_fields = 'pp'; % two scalar fields ("p" and "p")
vector_scalar_fields = 'up'; % a vector and a scalar field ("u" and "p")
dx = 2. * pi / 1024;
filterwidth = 7. * dx;
spacing = 4. * dx;

% for thresholding
threshold_field = 'vorticity';
```

```
threshold = 110.0;
X = int32(0);
Y = int32(0);
Z = int32(0);
Xwidth = int32(16);
Ywidth = int32(16);
Zwidth = int32(16);

points = zeros(3,npoints);
result1  = zeros(npoints);
result3  = zeros(3,npoints);
result4  = zeros(4,npoints);
result6  = zeros(6,npoints);
result9  = zeros(9,npoints);
result18 = zeros(18,npoints);

%  Set spatial locations to sample
for p = 1:npoints
  points(1,p) = 0.20 * p;
  points(2,p) = 0.50 * p;
  points(3,p) = 0.15 * p;
end

% Get data

vel_data = [];
spacing = 2.0*pi/1023;
for j = 1:100
     % Set domain size and position
     nx = 64;
     ny = nx;
     xoff = 2*pi*rand;
     yoff = 2*pi*rand;
     zoff = 2*pi*0.1;
     npoints = nx*ny;
     time = 0.002*j; % * randi(1024,1);
     clear points;

     % Create surface
     x = linspace(0, (nx-1)*spacing, nx);% + xoff;
     y = linspace(0, (ny-1)*spacing, ny);% + yoff;
     [X Y] = meshgrid(x, y);
     points(1:2,:) = [X(:)'; Y(:)'];
     points(3,:) = zoff;
```

```
    % Get the velocity at each point
    %fprintf('\nRequesting velocity at %i points...\n',npoints);
    fprintf('\nRequesting velocity at (%ix%i) points for velocity contour plot...\n',nx,ny);
    result3 = getVelocity(authkey, dataset, time, Lag4, NoTInt, npoints, points);

    % Calculate velocity magnitude
    z = sqrt(result3(1,:).^2 + result3(2,:).^2 + result3(3,:).^2);
    z = z.';
    vel_data = [vel_data, z];
    Z = transpose(reshape(z, nx, ny));
end

pressure_data = [];
spacing = 2.0*pi/1023;
for j = 1:20
   % Set domain size and position
   nx = 64;
   ny = nx;
   xoff = 2*pi*rand;
   yoff = 2*pi*rand;
   zoff = 2*pi*0.2;
   npoints = nx*ny;
   time = 0.002*j; % * randi(1024,1);
   clear points;

   % Create surface
   x = linspace(0, (nx-1)*spacing, nx);% + xoff;
   y = linspace(0, (ny-1)*spacing, ny);% + yoff;
   [X Y] = meshgrid(x, y);
   points(1:2,:) = [X(:).'; Y(:).'];
   points(3,:) = zoff;

   % Get the velocity at each point
   fprintf('\nRequesting velocity at %i points...\n',npoints);
   fprintf('\nRequesting velocity at (%ix%i) points for velocity contour plot...\n',nx,ny);
   result1 = getPressure(authkey, dataset, time, Lag4, NoTInt, npoints, points);

   % Calculate velocity magnitude
   z = result1;
   z = z.';
   pressure_data = [pressure_data, z];
   Z = transpose(reshape(result1, nx, ny));
```

```
   % manually saved vel and P data
end
```

This is function that performs DMD. It's from Kutz's DMD notes.

```
function [Phi,omega,lambda,b,Xdmd,S, U] = my_dmd(X1,X2,r,dt)
% Function copied from Kutz DMD Notes
% Computes the DMD of X1, X2

% DMD
[U, S, V] = svd(X1, 'econ');
r = min(r, size(U,2));

% Truncate to low rank
U_r = U(:, 1:r);
S_r = S(1:r, 1:r);
V_r = V(:, 1:r);
A_tilde = U_r' * X2 * V_r / S_r;
[W_r, D] = eig(A_tilde);
Phi = X2 * V_r / S_r * W_r;

lambda = diag(D);
omega = log(lambda)/dt;

% Compute DMD mode amplitudes
x1 =X1(:, 1);
b = Phi\x1;

% DMD reconstruction
mm1 = size(X1, 2);
time_dynamics = zeros(r, mm1);
t = (0:mm1 - 1)*dt;
for iter = 1:mm1
    time_dynamics(:, iter) = (b.*exp(omega*t(iter)));
end
Xdmd = Phi * time_dynamics;
```