Visualization of Vibration Experienced in Offshore Platforms

by

Alexander Marinos Charles Patrikalakis

S.B., Massachusetts Institute of Technology (2007)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2010
[June 2010]

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 24, 2010

Certified by .... [illegible] ....
Michael S. Triantafyllou
William I. Koch Professor of Marine Technology
Thesis Supervisor

Accepted by .... [illegible]
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Visualization of Vibration Experienced in Offshore Platforms

by

Alexander Marinos Charles Patrikalakis

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2010, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I design and evaluate methods to optimize the visualization of vortex-induced vibration (VIV) in marine risers. VIV is vibration experienced by marine risers in offshore drilling platforms due to ocean current flows, and appears to be perpendicular to the direction of such flows. VIV causes oil companies large capital losses, supply chain disruption, and environmental and brand name damage. For these reasons, both researchers and manufacturers try to improve their models of VIV, while creating risers more resilient to it. The first step to understanding VIV is rapid visualization, ie. the ability to efficiently visualize large amounts of simulated and field data. In this thesis, I evaluate high and low level heuristics that optimize the run-time performance of applications by taking advantage of 64-bit machines with large memory stores. Such heuristics include the introduction of object-oriented programming (OOP) with classes, dynamic binary loading, and source code management. I demonstrate that using these techniques allows speedups of many orders of magnitude, depending on the type of optimization and the structure of the input data. Finally, I reengineer an existing collection of disparate visualizations to take advantage of these heuristics, and achieve a run-time speedup of two orders of magnitude in most visualizations.

Thesis Supervisor: Michael S. Triantafyllou
Title: William I. Koch Professor of Marine Technology

Acknowledgments

I wish to thank my advisor Prof. Michael S. Triantafyllou for his guidance in arriving at and completing this thesis.

In addition, I wish to thank my academic advisor Prof. Jeffrey H. Lang for his guidance through my undergraduate and graduate studies at MIT.

I would like to thank my colleagues Haining Zheng, Fillipos Chasparis, Harish Mukundan, Yahya Modarres-Sadeghi, Rachel Price and Remi Bourget for their feedback on the VIV Suite visualizations.

I thank the Chevron and BP deep sea initiatives for funding this thesis.

Finally, I would like to thank my parents, Nicholas M. Patrikalakis and Sandra J. Patrikalakis; my brother, Andrew N. R. Patrikalakis; my sister, Nikki L. A. Patrikalakis; and, my friends, Georgios Papadopoulos, Yoshiaki Kuwata, Toshiyuki Takasaki, Taishi Nishida, Naoyuki Satoh, Kazuma Yokoh, Tracey K. Liu, and Samuel Dyar for their support during the creation and completion of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In this section, I introduce the basic concepts of marine risers, and the destructive physical phenomenon, known as vortex-induced vibration, that plagues these risers.

1.1.1 Marine Risers

A marine riser is a long, typically cylindrical, tube under tension that connects the hydrocarbon well-head on the ocean floor with the drilling or production platform at the surface of the ocean. Both crude oil and drilling equipment pass up and down marine risers. The length of marine risers in use today ranges anywhere from 100 meters (shallow-sea installations) to 5000 meters (ultra-deep sea installations). As it is difficult to create a single pipe that is so long, risers extend downwards towards the ocean floor, and pipe sections of varying length are fed through the top of the platform to elongate the existing riser. Each pipe section may have similar or different structural properties. For example, the existence of strakes (fins), the angle at which strakes are attached, the spacing of strakes, pipe thickness, and inner and outer radii are examples of structural properties that can differ among each section. A riser may also contain a combination of straked and bare sections, only bare sections, or only straked sections. A marine riser may be also partially covered by buoyancy modules reducing its effective weight in water and in turn diminishing the required magnitude of tension that needs to be applied on the riser from the offshore platform hydraulic system.

1.1.2 Vortex Induced Vibration

Vortex induced vibrations are an important design consideration for offshore platforms that extract hydrocarbons from well-heads on the ocean floor and transport them through risers to the ocean surface production offshore platforms, because the risers attached to such platforms are subject to large amounts of mechanical stress. One source of mechanical stress comes from the flow of sea water around such marine risers. The risers get in the way of the flow of water, and flow patterns known as vortices created in the wake of such flows cause the risers to vibrate, leading to substantial strain on them [3, 8, 11, 13]. These vibrations may cause fatigue damage and render the risers useless, so it is prudent to design the risers so that these vibrations are reduced.

Bluff solid bodies placed in a flow of fluid matter, such as liquid or gas, may cause the fluid stream to shed vortices (areas in the wake of a flow where the fluid appears to twist around itself) [9]. These vortices create alternating areas of low and high pressure behind the solid, causing the solid to vibrate in a direction perpendicular to the direction of fluid flow. These vibrations are known as vortex-induced vibrations. The risers attached to platforms are a good example of bluff objects subjected to vortex induced vibrations. Attaching strakes, or long helical fins, to the surface of risers has the effect of reducing the vibration response of risers to constant fluid flow [12].

VIV in marine risers is a real problem for oil companies that build and maintain offshore platforms for five reasons. First, damage to a marine riser represents a one-time capital loss for the oil company. Second, the loss of a marine riser to VIV-induced damage lowers the production capacity of the offshore platform in question, disrupting the supply chain of oil crude in all production streams. Third, a structural break in a marine riser also means that crude oil leaks into the ocean, causing environmental damage [7]. Next, the oil company is liable for unlimited cleanup costs and reparations up to 75 million dollars in damages to adversely affected industries, such as the fishing and tourism industries [5]. Finally, the oil company suffers a heavy blow to their brand value as a result of the environmental damage caused by riser failure [4]. For these four reasons, around ten percent of the cost of constructing an offshore platform is dedicated to countering the effects of VIV.

1.2 Research Objectives

The objective of this thesis is to demonstrate ways to make the visualization and analysis of VIV in marine risers more efficient and automatic. Taking advantage of the vast amounts of memory offered by modern 64-bit machines, as well as introducing modern software engineering practices to VIV visualization software, such as object oriented programming (OOP), dynamic library loading, and source code management all improve the run-time performance of programs. Specifically, I demonstrate how these optimizations improve the performance of VIV Suite, a suite of visualizations for VIV.

1.3 Thesis Outline

I provide an overview of this thesis in Chapter 1. In Chapter 2, I analyze the data domain of VIV events and analyze the information content of these events. In Chapter 3, I give a system overview of the VIV Visualization Suite, describing the class hierarchy, module interaction, external dependencies, development environment, and the Extract-Transform-Load (ETL) aspect of VIV events. In Chapter 4, I describe the data structures and abstract data types (ADT) used in VIV Suite, including VIV events in VIV experiments, VIVA simulator configuration files, and iterators. In Chapter 5, I describe the visualization and plotting algorithms developed for VIV Suite. In Chapter 6, I perform a design analysis of the improvements made to the unorganized collection of programs that was the predecessor of VIV Suite. Finally, I conclude in Chapter 7, by summarizing my contributions to VIV visualization methodology and giving directions for future work.

Chapter 2

Domain Analysis

In this chapter, I describe the data domain of VIV events.

2.1 VIV Experiment Structure

Researchers obtain experimental VIV data from two broad categories of experiments: field experiments and controlled experiments.

2.1.1 Field Experiments

Field experiments are carried out in ocean environments and allow researchers to observe outcomes in a natural setting rather than in a contrived laboratory environment. However, there are more variables and effects to consider.

2.1.2 Controlled Experiments

Controlled experiments are carried out in research laboratories, where experimental conditions, such as flow profile and the tension at the riser ends, can be controlled with more precision and accuracy.

2.1.3 Classification of Currently Available Data Sets

A classification of the data sets currently supported (processable) by VIV suite is listed in Table A.2.

2.2 Instrumentation

In both field and controlled experiments, a variety of instrumentation is placed along the span of a marine riser. They measure strain and acceleration from which other useful signals, such as curvature and displacement, can be extrapolated. Flow meters, if employed, measure fluid flow (current velocity and direction) along the span of the riser.

2.2.1 Strain Gauges

Strain gauges measure the strain experienced by a riser at certain points along the span of a riser.

Dimensional Analysis

Strain is a nondimensional physical quantity. Strain measures deformation in rigid bodies defined as the elongation caused by the application of external forces divided by the original length. As the order of magnitude of strain due to VIV in marine risers is often around 10^{-4}, some visualizations use a quantity known as microstrain. An example of an experiment where the strain is represented as microstrain to maintain the precision of the mantissa of floating point numbers is the NDP riser experiments [1]. Microstrain is 10^{-6} times smaller than a standard unit of strain.

A related quantity, curvature, is a dimensional physical quantity that has units $1/L$. Curvature also measures deformation of rigid bodies.

Configuration

Strain gauges are placed at equal or unequal intervals along the span of the riser. Often, multiple gauges of the same type, with different orientation, are placed at the same locations on the riser [2]; these gauges may measure cross-flow (CF) and in-line (IL) strain. Here, IL means in the direction of the ocean current that causes the vibrations, and CF means in the transverse direction. Experimental configurations with unequal numbers of CF and IL strain gauges are unusual but not unheard of [1].

Output

Strain gauges output a time series of strain measurements and writes these samples to text files. These measurements usually need to be adjusted for the calibration parameters of the gauges.

2.2.2 Accelerometers

Accelerometers measure the cross-flow and in-line accelerations, and by extension, the forces experienced by marine risers as a result of VIV.

Dimensional Analysis

Acceleration is a dimensional physical quantity that has units listed in equation 2.1.

$$\frac{L}{T^2} \tag{2.1}$$

Acceleration is the rate of change of velocity of a moving body. In the case of VIV events, acceleration refers to the rate of change of velocity of a riser subject to some flow. The SI unit of acceleration is meters per second squared.

Configuration

Accelerometers are placed at equal or unequal intervals along the span of the riser. Usually, an accelerometer is capable of producing three readings per sample, one for each dimension of three dimensional space. Thus, they are configured to have one dimension (z) parallel to the length of the riser, and two dimensions (x and y) perpendicular to the riser. Experimental configurations with unequal numbers of accelerometers in the x, y, and z directions are rare. Thus, references such as the "13^{th} accelerometer" usually refer to the collection of x, y, and z accelerometers, located at the 13^{th} accelerometer position.

Output

Accelerometers output a time series of acceleration measurements in three dimensions and writes these samples to text files. These measurements usually need to be adjusted for the calibration parameters of the accelerometers. Additionally, to account for any shifting and rotation, the acceleration time series in the x and y directions sometimes need to be rotated

2.2.3 Flow Meters

Flow meters measure the velocity of fluid flows across marine risers. They are usually used in field experiments [6, 14]; in controlled experiments, relative flow velocity is determined by keeping water stationary and measuring the motion of the riser relative to the fluid [2, 1].

Dimensional Analysis

Flow meters measure the velocity of the flow of a fluid – a dimensional physical quantity that has units listed in Equation 2.2.

$$\frac{L}{T} \tag{2.2}$$

Velocity is the rate of change of position of a moving object, and in the case of VIV events, flow velocity is the rate of change of position of fluid in flow across a marine riser. The SI unit of velocity is meters per second.

Configuration

Flow meters are usually placed at regular intervals along the span of a riser. Flow profiles are sampled less frequently than acceleration or strain signals. Often, there is only one flow profile that is supposed to be valid over the course of an entire event.

Output

Over the course of an event, each flow meter produces a time series of velocity readings. The readings are put through a digital-analog converter, scaled to the proper units, and saved to disk on a computer. Often, this time series does not change significantly over the life of a VIV event, so one frequently computes the average velocity of the in-line flow and treats it as a constant.

2.3 Information Content of Experimental VIV Data

The Norwegian Deepwater Program (NDP) conducted many experiments, known as cases, on bare and straked risers in a tow tank, subject to uniform and sheared fluid flow. Accelerometers and strain gauges generate approximately 80 time signals with order 104 number of samples each; in general, the NDP cases contain on the order of 106   107 floating point numbers. For example, NDP case 2430 contains 81 time signals of 30117 32-bit floating point samples each, plus 110 floating point numbers about the experimental setup, equaling 2.44 million 32-bit floating point numbers for that case. Without compression, one would expect 2.44 million 32-bit floats to consume 9530kB of disk space. Knowledge about the invariant sample frequency of the time vector signal in NDP case 2430 data yields the following intuition: having a constant sampling rate means that the time vector could be compressed from 30117 floats to exactly one integer (the number of samples) and two floats (sample interval and start time). Exploiting this knowledge allows the size of the data set to be reduced by 118kB to 9412kB. Furthermore, if the starting time is always zero, then the time vector could be represented as exactly one integer and one float.

This 'compression' trick assumes specific knowledge of the location of the time signal vector in the data set, and also assumes that the sample rate is constant for the duration of sampling. In other words, this compression technique depends on the specific conditions of a VIV experiment, and is definitely not portable across different sets of experiments, in which the data format is usually not the same.

Choosing compression methods that do not depend on meta-information concerning the data set obviously serve the interests of code portability across different data sets. For example, strain and acceleration data compose the overwhelming majority of information content in the NDP cases. The sensors that measure this data have physical limits concerning the precision, accuracy, and valid range of their measurements. In particular, while the precision and range of strain and acceleration data are parts of the physical limits of the instruments used for their measurement, the experimentally demonstrated subset of the instrumentation's precision and range can also be empirically obtained by statistical analysis of measured data.

Chapter 3

Design Description

In this chapter, I describe the overall design of VIV Suite, an optimized collection of visualizations for VIV.

3.1 System Overview

VIV Suite is largely a MATLAB application, with small C library and Fortran executable dependencies. The output of the VIVA simulator, a VIV simulator implemented in Fortran, according to Triantafyllou's model of VIV [10, 13], is at the heart of many of the simulations listed below.

1. Scalograms - Scalograms are three-dimensional surface plots of the time evolution of the frequency content of a time series.

2. Reconstructions - VIV events that meet the spatial frequency Nyquist criterion [8] postulated in Mukundan's doctoral thesis can be reconstructed at a very high spatial resolution (500 virtual accelerometers and strain gauges distributed

equally along the span of a riser). VISCO is a visualization that generates and visualizes this reconstruction.

3. Chaotic Analysis - The chaos visualization uses a gradient method among all CF acceleration signals of a VIV event to create a contour plot showing where the VIV response is chaotic, and where it is steady-state. This contour plot is a function of time and span.

4. Natural Frequency Analysis - This visualization uses mean-squared spectra to collect the frequency content of an acceleration or strain oscillation into discrete frequency buckets, in an effort to understand the aggregate response around specific natural frequencies. These natural frequencies are one of the outputs of the VIVA simulator.

5. Simulation Visualization with VIVOS - VIVOS compares the spanwise displacement RMS of real VIV events with the RMS generated by the VIVA simulator, as well as the simulated and real dominant frequencies.

6. Simulation Visualization with Movies - The movie generator adds random uniform noise around each of the natural frequencies generated by VIVA, generates sinusoids given these frequencies and their corresponding complex amplitudes, and animates these sinusoids as a function of span.

These visualizations are discussed in detail in Section 5.2. All of the visualizations use a common domain model to represent VIV events, which are described in Chapter 4, and render their output plots with the programmatic plotting API, which is described in Section 5.1.

3.2 Data Sources

Corporate sponsors such as oil companies provide data to us in a number of formats, including text files and binary storage formats, such as Microsoft Excel. Sometimes the dimensionality of the raw data is different depending on the experiment. For example, at times the readings for each strain gauge on a riser is in a different file, while other times, all the readings are in the same file.

3.3 Development and Deployment Platforms

The VIV Suite of applications runs on many platforms, and I used many different tools to develop and deploy these applications.

3.3.1 Development Platform

To achieve cross-platform portability, the majority of programs in the VIV Suite are currently written in MATLAB, with a few exceptions.

3.3.2 Back-End Tools

Critical sections of code are implemented in C++, linked with the MATLAB MEX Library, and dynamically loaded at runtime. Currently, the VIV Suite relies on a static dependency to the IPP Signal Processing libraries. Dynamic libraries for critical sections of code need to be compiled and linked for each host architecture that MATLAB runs on, namely 'MACI', 'GLNX', and 'PCWIN'. Finally, to maintain consistent execution results across multiple platforms, the VIVA programs should be compiled with ifort, Intel's Fortran compiler. Likewise, Fortran executables need to be created for each architecture that the VIVA simulator will be run on.

3.3.3 Front-End Tools

Currently, I use MATLAB to display all 2D graphs that constitute the output of the programs in the VIV Suite and for the static display of 3D graphs. I then use Apple QuickTime to encode, display, and store animations of 2D and 3D graphs.

3.3.4 Source Code Management

One of the large issues that I faced with the previous iteration of some of the programs in this suite was that instead of generalizing algorithms or customizing existing interfaces to deal with new data structures, my predecessors would make a copy of each program and custom-tailor it to the needs of each different data set. This ad-hoc source code management style caused the following two impediments to improving existing visualizations and adding new visualizations.

1. Multiple copies of the same program made it difficult to keep track of the most current version of a program.

2. The extent of the domain model of a VIV event was never clear from a single instance of a program; one had to examine the union of all VIV event properties available in all instances of the same program.

3. The lack of a historical record of execution scripts made it difficult to reproduce prior visualization results with complete confidence.

To alleviate the above issues caused by ad-hoc source code management, the first thing I did was to set up an SVN repository to manage one 'true' current copy of the VIV Suite. I then continued by adding a separate package for experiment-specific data extraction and transformation code; I discuss this package in Section 3.4.

3.4 Extraction, Transformation, and Load

Experimental VIV event data, usually provided by private corporations, does not adhere to any single storage standard. Thus, a large part of the experiment-specific code maintained in the experiments package deals with extracting, transforming, and loading (ETL) experimental VIV data.

3.4.1 Extraction and Transformation

The extraction and transformation stages for each experimental data set are implemented in MATLAB, and vary on the format the experimental data is provided in. As raw data is often provided in many tab-separated values (TSV) or comma-separated values (CSV) plain text files, one can use the import wizard in MATLAB to generate a program that will load one of these files, and then, to create one MATLAB data file (.mat) per VIV event, one needs to iterate over all the CSV/TSV files, change the normal form (for example, merging n strain time series from n files into one strain data matrix) of the data, rotate the data matrices, and finally, save the resulting vectors and matrices as .mat files.

3.4.2 Loading with Iterators

Loading individual VIV events from data files is done by implementing an iterator. Iterators abstract away the file operations necessary to load a VIV event.

Chapter 4

Data Structures and Abstract Data Types

In this chapter, I define the data structures (VivEvent, VIVA input file record types) and abstract data types (Iterator) used in VIV Suite. As the implementations of all of these data structures are MATLAB classes, null fields are represented by the empty array `[]`. Also, to avoid unnecessary data copying, all data structures and abstract data types inherit from the MATLAB `handle` class, allowing for mutable objects dereferenceable by handles.

4.1 VivEvent

The VivEvent class stores experimental data from a VIV event. It provides a structure for maintaining data about an experiment in the same place, and at the same time it provides a consistent interface to data processing clients, as it mandates a uniform nomenclature for all fields.

4.1.1 Base Fields and Their Representation Invariants

VIV event fields can be divided into categories such as riser properties, event records, hydrodynamic properties, and event metadata. In the following sections, I will discuss fields of each of these types.

Riser Properties

1. `diameter` - This is the diameter of the riser in a VIV event in meters.
2. `riserLength` - This is the length of the riser in a VIV event in meters.
3. `CFaccelz` / `ILaccelz` - These are vectors containing the positions of the accelerometers along the span of a riser in meters, in the CF and IL directions. Thus, the length of one of these vectors is equal to the number of accelerometers in a VIV event in a particular direction.
4. `CFstrainz` / `ILstrainz` - These are vectors containing the positions of the strain gauges along the span of a riser in meters, in the CF and IL directions. Thus, the length of one of these vectors is equal to the number of strain gauges in a VIV event in a particular direction.
5. `CFdisplz` / `ILdisplz` - These are vectors containing the positions of the displacement measurements along the span of a riser in meters, in the CF and IL directions. Thus, the length of one of these vectors is equal to the number of displacement measurements in a VIV event in a particular direction.

Event Records

1. `CFaccln` / `ILaccln` - These matrices, with the number of rows equal to the number of accelerometers in each direction, and the number of columns equal to

the length of the time vector, are the acceleration signals as measured at each accelerometer, in each direction.

2. `CFstrain` / `ILstrain` - These matrices, with the number of rows equal to the number of strain gauges in each direction, and the number of columns equal to the length of the time vector, are the strain signals as measured at each strain gauge, in each direction.

3. `CFdispl` / `ILdispl` - These matrices, with the number of rows equal to the number of accelerometers in each direction, and the number of columns equal to the length of the time vector, are the displacement signals as interpolated from the accelerations measured at each accelerometer, in each direction.

4. `time` - This vector is the time signal of the VIV event.

Hydrodynamic Properties

1. `velz` - This vector is a vector of the absolute positions of the flow meters along the span of the riser.

2. `velocity` - This vector represents the flow profile of a VIV event, and is the average velocity of flow in the IL direction for the duration of the event.

Event Metadata

1. `expname` - The `expname` field is a string representing the name of the experiment. This string may not be empty. This string is used programmatically to generate paths to experiment data, so it must be the name of an actual directory in the `experiments` directory of the VIV Suite root directory.

2. `eventno` - The `eventno` field is an integer representing the event number in a set of VIV events belonging to an experiment. This number is not padded with zeros on the left, and is used to generate paths to event data and output files.

3. `partno` - The `partno` field is an integer representing the part number of a VIV event instance. Using the PartsIterator, a VIV event can be split into multiple parts, so that they can be visualized individually. The field `partno` may vary from 1 to `parts`.

4. `parts` - The `parts` field is an integer representing the total number of parts of a VIV event. These parts are VIV events themselves, and are generated from a whole VIV event using the `PartsIterator` class.

4.2 VIVA Input File Record Types

The VIVA simulator requires a number of files as input to define the parameters of the riser and the flow profile used in a simulation [13]. Previously, these files were generated by scripts that wrote each configuration file line by line. Such scripts were difficult to understand and error-prone because each was simply a list of numbers, with no apparent structure. They were error-prone because, as a linear script, no constraints were enforced on the contents of the script. Specifically, lines could be missing, or alternatively, there could be extra lines in the configuration files, and no one would be the wiser until the VIVA programs were run and crashed because of poorly-formed configuration files.

The record types corresponding to the VIVA input files are the following (their fields are defined in detail in the VIVA manual [13]).

1. Riser Dynamics - risdyn-n.in to specify the ocean current data.

2. Riser Fatigue - risfat.in to specify the fatigue curves.
3. Riser Preferences - rispre.in to specify the riser data.
4. VIVA Conditions - conditions.in to specify the boundary conditions.

4.3 Iterators

The `Iterator` class follows the Iterator design pattern in that it presents a uniform way to iterate over a number of VivEvents. The Iterator is not traditional as it does not iterate over the contents of a concrete container of VivEvent objects; rather, it iterates over an abstract collection of VivEvents that are stored on disk.

`Iterator` implementation maps an experiment's data files containing VIV events to an abstract, lazy-loading container that instantiates VivEvent objects only when they are iterated to. The cost of RAM makes maintaining a collection of all an experiment's VivEvents in RAM infeasibile, thereby creating the need for lazy-loading containers. In addition to optimizing system memory usage, the concrete `Iterator` subclasses encapsulate code that extracts and transforms the VIV data contained in the flat text files provided by the oil companies. In other words, the concrete Iterator subclasses encapsulate the Extract-Transform-Load (ETL) logic specific to each experiment. Thus, strategy objects that use an `Iterator` to create a visualization of the VIV contained within each event no longer need be aware of how VIV data gets loaded and transformed from files.

4.3.1 Iterator Interface

Each concrete `Iterator` subclass needs to implement the following instance methods.

1. `y = next(obj)` - The `next` method of a concrete Iterator implementation must return a handle to the next VIVEvent object. The prerequisite of this method is that the `hasNext` method of the iterator must return true. If `hasNext` does not return true, then the call to `next` will fail with an error.

2. `pathStr = generatePathToEvent(obj, eventNo)` - The `generatePathToEvent` method is given an event number and generates a file path string to the MATLAB data file that contains the data necessary to construct a VIVEvent object for the VIV event indicated by `eventNo`.

3. `n = getLength(obj)` - The `getLength` method takes no arguments and returns the number of VIV events able to be iterated to by using a particular instance of a concrete Iterator implementation.

4.3.2 Implementing Subclasses

There are two kinds of Iterator implementations. The first kind are the concrete iterators that have a one-to-one relationship with the data format of each VIV data set. The second kind is called the `PartsIterator`, which can divide one `VivEvent` instance into multiple `VivEvents`, given an array of times at which to divide the VivEvent. The `PartsIterator` can handle `VivEvents` generated by any concrete `Iterator` subclass.

Chapter 5

Algorithms

This chapter describes the rendering methodology and specific visualizations implemented in VIV Suite.

5.1 Plotting API

The plotting application programming interface (API) is used by all visualizations in VIV Suite to render the plots they output. The API enables the batch rendering of output plots by enabling their intermediate representation as strategy object instances that describe how exactly to draw plots. The API also allows visualizations to consistently reproduce identical visualization results because plot composition is defined in a declarative fashion at compile time, rather than in a descriptive fashion (as a quick-and-dirty script) at run time. Finally, as the API abstracts the view layer of VIV Suite away from the rest of the application code, migrating VIV Suite to another language such as C++ becomes easier because the MATLAB-specific subroutine dependencies are walled off in one place.

5.1.1 Plotters

Each plot type that is supported by MATLAB is implemented as a subclass of the Plotter abstract class. Plotters implement UI elements common to all plots: x-axis labels, y-axis labels, z-axis labels, titles, x-axis limits, y-axis limits, and z-axis limits. However, all of these elements are optional; for example, if one does not want to have an x-axis label, one can simply put the empty string in the x-axis label string. The Plotter superclass intentionally leaves representation details unspecified, such as the method of storing plottable data to give maximum abstraction flexibility to the implementing subclasses.

The Plotter subclasses currently implemented are the following:

1. `ImageScalePlotter` - The `ImageScalePlotter` plotter is based on the MATLAB `imagesc` plot and draws bitmaps to the screen. An example of the `ImageScalePlotter` can be seen in the reconstruction visualization output described in Section 5.2.3.

2. `LandscapePlotter` - The `LandscapePlotter` plotter is based on MATLAB's call to set the orientation of a plot to landscape mode. This plotter must be run in sequence with other plotters, or alternatively, requires that a figure be open and active. The `LandscapePlotter` plotter can be seen in action in the VIVOS simulation visualization output described in Section 5.2.4.

3. `LinePlotter` - The `LinePlotter` plotter is based on MATLAB's `plot3` plot and draws a line in three-dimensional space, given a sequence of three-dimensional coordinate pairs. An example of the `LinePlotter` can be seen in the scalogram visualization output at the front-most edge of the three-dimensional figure described in Section 5.2.2.

4. `MSSPlotter` - The `MSSPlotter` plotter is based on MATLAB's `MSSpectrum` plot and draws the mean-squared spectrum of a time series. An example of the `MSS-Plotter` can be seen in the natural frequency visualization output described in Section 5.2.1.

5. `RegularPlotter` - The `RegularPlotter` plotter is based on MATLAB's `plot` plot and draws the plot of a two-dimensional signal, given a vector of x-coordinates and a vector of y-coordinates. An example of the `RegularPlotter` can be seen in the simulation visualization output; the flow profiles in VIVOS described in Section 5.2.4 are drawn with this plotter.

6. `StemPlotter` - The `StemPlotter` plotter is based on MATLAB's `stem` plot and draws the plot of a two-dimensional signal as line bars, given a vector of x-coordinates and a vector of y-coordinates. An example of the `StemPlotter` can be seen in the natural frequency analysis output; the mean-squared spectra of acceleration and displacement described in Section 5.2.1 are drawn with this plotter.

7. `SurfacePlotter` - The `SurfacePlotter` plotter is based on MATLAB's `surf` plot and draws a three-dimensional surface, given a z matrix and x and y axis values (each in a vector). An example of the `SurfacePlotter` can be seen in the scalogram visualization output as the scalogram surface itself described in Section 5.2.2.

8. `TwoDFFTPlotter` - This plotter takes a matrix of time series, in row major order, and generates the two-dimensional Fourier transform of the matrix. This plotter is used to display the frequency-based and wave number-based spectral content of the reconstructed signals in the reconstruction visualization output described

in Section 5.2.3.

9. `VerticalLinePlotter` - The `VerticalLinePlotter` draws a vertical line on an existing plot. It requires that a figure already be open and active. This plotter is used to demarcate the frequency band used in the reconstruction visualization output described in Section 5.2.3.

5.1.2 Composite Plotters

The plotting API also includes some composite plotters that allow other plotters to be stitched together in various ways.

The CompositePlotter Class

The constructor of the `CompositePlotter` class takes as input a cell array of other Plotter subclass instances and runs each Plotter instance in succession on the same figure. The `CompositePlotter` class uses MATLAB's `hold on` and `hold off` functions to make multiple plots stick to the same figure. It can be seen in action in the VIVOS simulation visualization output (Section 5.2.4), where it allows the flow profile, simulated RMS, and experimental RMS to all be displayed on the same plot.

Page Spanner

The constructor of the `PageSpanner` class takes as input a cell array of other Plotter subclass instances along with the overarching x-axis label, left y-axis label, right y-axis label, and title for an array of plots. The `PageSpanner` class makes use of MATLAB's `subplot` function, as well as the external `suplabel` function. A `PageSpanner` automatically paginates and moves plots over to the next page if there are more Plotters in the cell array than can be displayed on one page. The `PageSpanner` can be seen in action in the VIVOS simulation visualization output (Section 5.2.4), where it allows the flow profile, simulated RMS, and experimental RMS for each VIV event to be displayed in a different coordinate system in the plot grid.

5.2 Visualizations

In this section, I discuss the numerous visualizations that VIV Suite implements. These include natural frequency analysis, scalograms, reconstructions, simulation visualization with VIVOS, simulation visualization with movies, and chaotic response analysis.

5.2.1 Natural Frequency Buckets

The `naturalFrequencies` function separates the frequency content of displacement in a VIV event, computed from acceleration, into groups centered on a number of natural frequencies.

Interface

```
function naturalFrequencies(expRoot, iter, sensorNumbers, band, naturalFreqsCells,
    selectedNaturalFreqs, outputFormats)
```

Usage

The string `expRoot` refers to the directory for a given experiment. This directory will be the parent directory of the files that are written by the naturalFrequencies function. The object `iter` is an Iterator that can access at least one VivEvent (in other words, the length of the iterator is at least 1). `sensorNumbers` is a vector of integers that enumerates the sensor identifiers the client wishes to process using the naturalFrequencies function. The two-element vector band indicates the low and high frequency bounds that will be displayed on all the plots that are written by this function. naturalFreqs is also a cell array of vectors of real numbers that enumerate the natural frequencies around which the function should cluster frequency content for each VIV event indicated by the iterator. The selectedNaturalFreqs parameter is a cell array containing vectors of the important natural frequencies of each VIV event indicated in the iterator iter, and the parameter outputFormats is a cell array of file extension strings corresponding to the file formats that the natural frequency output should be written to. This function requires the summ1.out and freq.out files generated by the VIVA simulator [13] to be available for each VIV event iterated to by iter. This function returns no value.

Output Directory Structure

The default output directory is ⟨expRoot⟩/OUTPUT/FREQ. Both the OUTPUT directory and the FREQ directory contained within will be generated by naturalFrequencies if they do not exist. The graphs for VIV events iterated through naturalFrequencies will be written to the FREQ directory.

Output File Structure

The function naturalFrequencies writes five plots to files for each VIV event iteration. In the following file names, ⟨event number⟩ stands for digits that correspond to a particular VIV event in an experiment, N is the page number of a particular collection of plots, and M is the total number of pages associated with a particular plot.

1. ⟨event number⟩accel_N_of_M.pdf - This file contains plots of the mean-squared spectrum of the acceleration signals measured at each of the accelerometers

along the span of the riser in the VIV event. In case there are more than 16 accelerometers, the maximum number of signals that can be displayed on a page, there will be multiple files, with N ranging from 1 to M. An example figure is listed in Figure B-6.

2. ⟨`event number`⟩`accelNatural_N_of_M.pdf` - This file contains plots of the natural frequency content of an acceleration signal, as computed from its mean squared spectrum. In case there are more than 16 accelerometers, the maximum number of signals that can be displayed on a page, there will be multiple files, with N ranging from 1 to M. An example figure is listed in Figure B-7.

3. ⟨`event number`⟩`displ_N_of_M.pdf` - This file contains plots of the natural frequency content of the displacement corresponding to each accelerometer in a VIV event. This displacement spectrum is obtained by scaling each power value by $(1/2\pi\omega)^4$, where omega is the frequency local to the power in question. This is the equivalent of integrating twice in the frequency domain. In case there are more than 16 accelerometers, the maximum number of signals that can be displayed on a page, there will be multiple files, with N ranging from 1 to M. An example figure is listed in Figure B-8.

4. ⟨`event number`⟩`displAmp_N_of_M.pdf` - This file contains plots of the natural frequency content of the displacement amplitude, corresponding to each accelerometer in a VIV event. An example figure is listed in Figure B-9. The displacement amplitude is related to the displacement listed in item 3 by the following equation:

$$\text{amplitude} = \sqrt{2} * (\text{displacement RMS}) \tag{5.1}$$

5.2.2 Scalograms

The `plotScalosIterant` function creates scalograms (3D time-frequency plots) at fixed points along the span of a riser, in a set of VIV events indexed by an Iterator. Scalograms are created by convolving Morlet wavelets, of varying length, with the Hilbert transform of a time series. Strain, stress, acceleration, or displacement are good representative examples of time series convolved with wavelets. Each Morlet wavelet corresponds to a particular frequency, and the successive frequencies decrease exponentially. The convolution product is called a frequency bucket and is a function of time, and if these frequency buckets are laid side by side in a 3D plot, it is referred to as a scalogram.

Interface

```
function plotScalosIterant(nframes, window, freq, cumPowerRatio, autoBanding,
    points, timeRange, iter, outputFormats, expRoot, voices, equidistantPoints)
```

Usage

The parameter `nframes` indicates the number of time slices the scalogram should be divided into. When `nframes` is greater than 1, a movie showing the scalogram growing to the maximum record length is generated instead of individual still frame images. The parameter `window` indicates the length of the sliding average window used to silence high frequency noise in the time domain of the time-frequency-response (TFR) matrix. Having a `window` of length 30 generally produces scalograms that are smooth. `freq` indicates the frequency band for which the scalogram should be computed. The cumulative power ratio, or `cumPowerRatio`, is a real scalar from 0 to 1 indicating the percent of mass of the TFR matrix that needs to be present to generate frequency bounds within `freq` automatically. For example, the figure 0.95 in place of the cumulative power ratio means that plotScalosIterant will display the frequency range that contains 95 percent of the volume of the TFR matrix. The parameter points indicates either an array of points if the boolean equidistantPoints is false, or the number of equidistant points along the span of the riser at which to calculate scalograms, if equidistantPoints is true. The iterator iter is an iterator object pointing to a collection of VIVEvents. outputFormats is a cell array of file extension strings, the formats in which scalograms will be written to disk. expRoot is the root directory of the experiment whose scalograms are being computed. voices represents the number of wavelets to be calculated when generating the scalogram. The larger voices is, the longer a scalogram takes to compute.

Output Directory Structure

The default output directory is ⟨expRoot⟩/OUTPUT/SCALO. Both the OUTPUT directory and the SCALO directory contained within will be generated by plotScalosIterant if they do not exist. The graphs for VIV events iterated through in the function plotScalosIterant will be written to the SCALO directory.

Output File Structure

plotScalosIterant generates files with each of the file extensions in the outputFormats cell array using the following naming convention. An example figure is listed in Figure B-10.

⟨expname⟩⟨config⟩⟨eventNum⟩C⟨sensorNum⟩S⟨signal⟩⟨window⟩W⟨frameNum⟩F.⟨ext⟩

5.2.3 Reconstructions

The `run_VISCO` function reconstructs a set of VIV events indexed by an Iterator. Here, reconstruction means the process of interpolating displacement and strain time series at many points along the length of the riser in a VIV event, provided that the sensor positions satisfy the spatial frequency criterion [8]. Harish Mukundan discusses the interpolation algorithm, and the spatial frequency criterion in his doctoral thesis.

Interface

```
function run_VISCO(iterIn, sensex, cutofffreq, summaryOnly, graphFormats,
    savingRecon, savingPSD, savingRMS, nodalRadius, exproot, useMicroStrain,
    starts)
```

Usage

The Iterator `iterIn` is an iterator that can access at least one `VivEvent` (the length of the iterator is at least 1). `sensex` is a list of failed sensor numbers that need to be excluded from the interpolation algorithm. The vector `cutofffreq` is a pair of frequencies, measured in Hz, representing the frequency band to be used in a band-pass filter applied to strain and acceleration signals in VIV events. When the boolean `summaryOnly` is false, summary (four plots in one page) and detailed plots for each of these four visualizations are written to disk; however, when `summaryOnly` is true, only the summary plots are written to disk. The cell array `graphFormats` contains a list of strings, corresponding to the desired file extensions for the output files. The boolean `savingRecon` determines whether the actual reconstructed signal will be saved to disk, and the boolean `savingPSD` determines whether the spectrum data of the span-averaged PSD will be saved to disk.

Output Directory Structure

The default output directory is ⟨`expRoot`⟩`/OUTPUT/VISCO`.

Output File Structure

The output of the `run_VISCO` function consists of:

1. `CF_Fourier_recon_disp_summ_N_of_M.pdf`, a summary plot of reconstructed displacement RMS with experimental displacement RMS values superimposed for each VIV event in the input iterator, An example figure is listed in Figure B-11;

2. `CF_Fourier_recon_strain_summ_N_of_M.pdf`, a summary plot of reconstructed strain RMS with experimental strain RMS values superimposed for each VIV event in the input iterator. An example figure is listed in Figure B-12;

3. a summary plot of the Fourier coefficients used in the reconstruction of each VIV event in the input iterator;

4. `summary_N_of_M.pdf`, a two-by-two summary plot of four visualizations of the reconstructed VIV event (nodal plot, 2D FFT, span-averaged PSD, and a sample displacement signal). An example figure is listed in Figure B-13; and

5. optionally, detailed plots of each of the visualizations of the previous item.

5.2.4 Simulation Visualization with VIVOS

The `run_VIVOS` function compares VIVA prediction of VIV amplitude, strain, and dominant frequency with experimental data from marine risers.

Interface

```
function run_VIVOS(diameter, riserLength, datacase, datacaseIndex, numDatacases,
    config, expname, outputFormats, vivosOutputLocalDir, vivosSummaryOutputDir,
    vivaOutputLocalDir, vivaInputLocalDir, rmsSummary, isStrain)
```

Usage

The numbers `diameter` and `riserLength` are the values of riser diameter and length, respectively. The number `datacase` refers to the datacase number, and the datacaseIndex refers to the index of the datacase inside the experiment. The number `numDatacases` indicates the total number of datacases in certain experiments. The string `config` refers to the the configuration of the datacase, such as whether it is with or without strakes and whether the flow is uniform or sheared. `expname` is the name of the experiment, and `outputFormats` is a cell array of file extension strings, the formats in which run_vivos will be written to disk. `vivosOutputLocalDir` is the directory where `run_VIVOS` will output the results. `vivosSummaryOutputDir` is inside `vivosOutputLocalDir`, and is the directory where a output summary is generated for quick review. The string `vivaOutputLocalDir` is the directory containing the VIVA prediction output, and the `vivaInputLocalDir` is the directory where basic_bare file is stored. The string `rmsSummary` contains the experimental results, and the boolean variable `isStrain` determines whether there is strain for amplitude analysis.

Output Directory Structure

The default output directory is ⟨`expRoot`⟩`/OUTPUT/VIVOS`, and the summary graphs will be written to the `VIVOS/ALL_SUMMARIES` directory.

Output File Structure

The function run_VIVOS writes three summary plots to files, namely the amplitude/strain comparison (An example figure is listed in Figure B-15), frequency comparison (An example figure is listed in Figure B-16), and basic_bare used by viva (An example figure is listed in Figure B-14). The `⟨basic_bare⟩.pdf` file visualizes the hydrodynamic database used to govern VIV in the VIVA simulator [10, 3]. A hydrodynamic database is represented by the following plots (in order from left to right and top to bottom):

1. **lift** - the lift coefficient in phase with velocity at $A/D = 0$;
2. **added-mass** - the corresponding added mass coefficient;
3. **first-slope** - the first slope of the lift curve;
4. **second-slope** - the second slope of the lift curve; and
5. **nondimensional amplitude** - is the value of A/D where the slope changes.

For the comparison figures, in the file names `⟨config⟩` stands for a particular VIV experiment configuration, N is the page number of a particular collection of plots, and M is the total number of pages associated with a particular plot.

1. `⟨config⟩_disp_rms_nom_N_of_M.pdf` indicates displacement comparison
2. `⟨config⟩_strain_rms_nom_N_of_M.pdf` indicates strain comparison
3. `⟨config⟩_freq1_nom.pdf` indicates frequency comparison.

The function run_VIVOS also writes three data files for each VIV event iterated through. `digidata_EXP_disp_⟨event number⟩.pdf` is the digital output of the experimental displacement/strain and `digidata_VIVA_disp_⟨event number⟩.pdf` is the digital output of the viva displacement/strain predication. The file `velocity.OUT` contains the velocity profiles.

5.2.5 Simulation Visualization with Movies

The `makeVIVMovie` function generates a movie of nominal VIV-based CF displacement in risers, given the frequencies and complex amplitudes of vibrations predicted by the VIVA simulator.

Interface

```
function makeVIVMovie(eventNumbers, periods, position, freqBounds,
    useDominantHarmonicOnly, expRoot, config, expname)
```

Usage

The vector `eventNumbers` shows the datacases in the experiment, `periods` is the number of cycles shown in the movie, `position` indicates the position on the riser where the movie is made, and `freqBounds` limits the frequencies included in the movie. The boolean variable `useDominantHarmonicOnly` determines whether only dominant harmonic is used. The string `config` refers to the configuration of VIV event, such as whether it is with or without strakes and whether the flow is uniform or sheared. `expname` is the name of the experiment.

Output Directory Structure

The default output directory is ⟨`expRoot`⟩`/OUTPUT/MOVIE`.

Output File Structure

The function `makeVIVMovie` writes one movie file and one summary file containing spectrum analysis of natural frequency for each VIV event iterated through. N is the page number of a particular collection of plots, and M is the total number of pages associated with a particular plot. An example figure is listed in Figure B-17

1. ⟨`casenumber.avi`⟩is the movie displaying the VIVA amplitude as a function of time;
2. ⟨`summary_casenumber`⟩ shows the complex amplitude Ψ from VIVA, the displacement time series, the reconstructed event spectrum with uniform random phase, and the probability of harmonics.

5.2.6 Response Analysis

The `Chaos_in_VIV` function categorizes the span-time plane of VIV response into two categories: steady-state response and chaotic response.

Interface

```
function Chaos_in_VIV(intervals, tolerance, iter, expRoot, outputFormats)
```

Usage

The string `intervals` is the number of time-axis subdivisions, while `tolerance` is a measure of how big a peak in the PSD plot should be to be considered comparable with the largest peak. Having a tolerance (`tol`) of 0.5 means that peaks of a height that is 50% of that of the main peak are considered comparable. The variable `iter` is an iterator to VIV events. The variable `expRoot` is the experiment root directory, and outputFormats is a cell array of output graph extensions. The string intervals is the number of time-axis subdivisions, while tolerance is a measure of how big a peak in the PSD plot should be to be considered comparable with the largest peak. Having a tolerance (tol) of 0.5 means that peaks of a height that is 50% of that of the main peak are considered comparable. The variable iter is an iterator to VIV events. The variable expRoot is the experiment root directory, and outputFormats is a cell array of output graph extensions.

Output Directory Structure

The default output directory is ⟨expRoot⟩/OUTPUT/CHAOS.

Output File Structure

The function makeVIVMovie writes one summary file Chaplin10Chaos for each VIV event that is iterated to; an example figure is shown in Figure B-18.

Chapter 6

Design Analysis

In this section, I will discuss the system, program, and statement-level optimization strategies employed by VIV Suite to improve visualization performance for VIV.

6.1 Optimization Strategies

There are a number of optimization strategies that I employed in the renewal of VIV Suite. System-level strategies optimize the performance of groups of programs or machines as a whole. Program-level optimization strategies deal with the performance of individual programs. Finally, statement-level strategies deal with optimizing the performance of programs at the statement level.

The specifications of the system I used to perform the experiments described in this chapter are listed in Table A.1.

6.1.1 System-level Strategies

System-level strategies orchestrate groups of machines, either connected by a network or by the bus on the same computer, with the goal of achieving high parallelization ratios. Some approaches do this by encapsulating entire programs in virtual machines, and others do this by hiding parallel computing facilities behind loop constructs.

Virtualization

Virtualization is one system-level optimization strategy. It consists of virtualizing an application, such as MATLAB, in a virtual machine, and subsequently deploying many copies of this virtual machine on host computers. Running multiple copies of a computer that does the same computation on different inputs increases the effective throughput of the computation; ideally, the increase is linear to the number of virtual machines. Virtualization does not reduce the latency of a particular computation, since one computation, with the same inputs and on the same computer, will always take the same amount of time, barring all other changes.

I used virtualization to increase the effective throughput of scalogram computation for NDP38 VIV events. I virtualized MATLAB in an Ubuntu 9.04 32-bit virtual machine, and had the scalogram code installed in the virtual machine. I made multiple copies of this machine and ran 10 of them on a total of 10 execution threads (cores) spread through 3 computers.

MATLAB Parallelization

MATLAB has parallel execution paradigms, such as a parallel for loop, that allow it to automatically parallelize programs for multiple cores. However, utilizing these parallel constructs requires purchasing the Parallel Computation Toolbox, and thus I opted not to try this route.

6.1.2 Program-level Strategies

Program level optimization strategies seek to optimize performance in time and space at the level of individual programs and modules in these programs. File IO and rendering are examples of program aspects that can be optimized.

Render Time Amortization

Sometimes legacy programs will compute data to be plotted and subsequently plot the data in parts, since the available system memory limits the amount of data that can be plotted simultaneously. Thus, these programs would divide the computation into equal parts and iterate through each part, computing and plotting each result in sequence. With the advent of 64-bit machines, these memory considerations are no longer an issue, so it is conceivable that computing and then plotting in batch may be more efficient in time.

To validate my claim, I performed an experiment in which I generated and plotted 1, 4, 9, 16, 25, 36, and 49 plots of distinct sequences of 10000 uniformly distributed random numbers. First, I generated data and plotted graphs interactively, intertwining data generation and plot rendering. Second, I batched the computation and plotting stages. In both cases, I measured the time taken to process as a function of the number of subplots.

I found that when the number of subplots is on the order of 10 subplots or less, computing and plotting in batch performs significantly better over time than interactive plotting does, and that for larger numbers of subplots, computing and plotting in batch performs marginally better than interactive plotting does. The results of this experiment are shown in Figure B-1.

As many visualizations in VIV Suite generate plots that have $O(10)$ or less subplots, I was able to improve the time performance of many visualizations with this technique.

File I/O Amortization

Again, past legacy applications had to deal with memory limitations that sometimes did not allow them to keep all their intermediate variables and plots in memory for a particular computation. 64-bit computers now allow process images to consume more memory resources, so reducing the time cost of file I/O by rendering and writing all plots in batch is possible.

I tested the following two approaches to amortize or eliminate the cost of file I/O in VIV Suite.

1. **Batch output figure writes** - To validate my claim that batch disk writes are more efficient than staged writes, I performed an experiment in which I generated, plotted, and saved 1, 4, 9, 16, 25, 36, and 49 plots of distinct sequences of 10000 uniformly distributed random numbers. First, I interactively generated, plotted, saved, and closed the figure with each successive subplot that was added, loading the intermediately saved figure before each new subplot. Second, I generated all the data to be plotted, plotted it in batch, and finally saved the resulting plot to disk once. In both cases, I measured the time taken to process the plots and save them to disk as a function of the number of subplots.

   I found that for any number subplots, plotting in batch and saving once results in time performance increases of 1 to 2 orders of magnitude as compared to plotting and saving figures in stages. The results of this experiment are shown in Figure B-2.

The VISCO visualization, which generates Mukundan's signal reconstruction [8], and the VIVOS visualization, which compares experimental vibration RMS data to simulated vibration RMS data, both deal with large numbers of subplots as they generate their outputs. Both of these visualizations benefitted from the amortization of file I/O, and became 10 to 100 times faster, as demonstrated by my simple experiment.

2. **Eliminating the use of temporary files** - Legacy systems that have small amounts of physical memory sometimes would not be able to hold the entire state of an application in memory. Sometimes, instead of letting the VMM paginate the portion of application state that does not fit in real memory, the application intentionally persists all variables that are part of a frame to disk before a frame switch, performs the frame switch, and then loads the variables necessary for the new frame from disk. In this case, a frame switch means a function, or instance method, call in MATLAB.

   I performed an experiment in which I passed an array of floating point numbers of increasing length to another function. I compared the time it took to pass this array by value as a function parameter to the time it took to pass this array through a temporary file on disk, as I increased the length of this array from 10 numbers to $10,000$ numbers.

   I found that passing arrays (and by extension, matrices) of any size by value is consistently one order of magnitude faster than writing the array to a temporary file on disk, and subsequently reading it in the client function. The results of this experiment are shown in Figure B-3.

   All of the previous visualizations used temporary variables to store their intermediate application state and to communicate data between functions, as the computers they were run on did not have enough memory to store all application state in the MATLAB process image at the same time. By eliminating the use of temporary files to pass data between functions, I have made frame switches in VIV Suite one order of magnitude faster.

3. **Passing variables between methods by handles, not by value** - MATLAB handle objects are like references in Java and pointers in C/C++. They represent a non-decreasing integer sequence that is unique for each class, and each value in this sequence addresses a specific instance of the class, as they are created. Passing object handles between functions and not between plain arrays and matrices reduces the time needed to switch into a new function frame from an amount that is proportional to the size of the parameters to an amount that is proportional to the number of parameters, which is effectively constant.

   I performed an experiment in which I pass an array of floating point numbers of increasing length to another function. I compare the time it takes to pass this array by value as a function parameter, to the time it takes to pass this array as member state of a handle class instance, as I increase the length of this array from 100 numbers to $10,000,000$ floating point numbers (the equivalent of 40 MB of data, roughly the size of a large VIV event on disk).

   I found that passing arrays (and by extension, matrices) up to $20,000$ floating point numbers in size by value is consistently one order of magnitude faster than passing the array by using handles. However, for arrays that are greater than $20,000$ floating point numbers in size, the frame switch time using handle classes is bounded by $100 \quad \mu s$, whereas frame switch time when passing variables by value is linearly proportional to the size of the array, and thus unbounded. The results of this experiment are shown in Figure B-4.

Passing OOP class instances between methods enables frame switches in MATLAB in constant time. I introduced object orientation to all parts of VIV Suite, and thereby reduced the time spent in frame switches within VIV Suite functions to $O(1)$.

6.1.3 Statement-level Strategies

Finally, I will discuss statement level optimization strategies including static array allocation and the delegation of critical code sections to external programs. Sometimes these programs are independent applications that are implemented in C or Fortran; other times, they are dynamically loaded libraries that get run from within the MATLAB process image.

Static Array Allocation

MATLAB allows programmers to define variables on the fly. Not only can undeclared variables be referred to (and thus implicitly declared), they can also be resized, by means of recursive assignment, on the fly. Listing 6.1 shows an example of how to statically preallocate all variable storage space needed *before* entering a for loop.

Listing 6.1: Static Array Allocation

```
myzeros = zeros(5,5);
for i=1:5
    myzeros(i,:) = [0,0,0,0,0];
end
```

In contrast, Listing 6.2 shows how one can dynamically allocate an array in MATLAB. This example appends a row of zeros to the extant `myzeros` variable.

Listing 6.2: Dynamic Array Allocation

```
myzeros = [];
for i=1:5
    myzeros = [myzeros; 0,0,0,0,0];
end
```

Dynamic allocation of arrays, as shown in Listing 6.2 is inefficient because the `myzeros` array has to be copied to a temporary store before being overwritten, so appending data incurs the same time cost as passing this array *by value* (see Section 6.1.2) to a function that does the appending. Therefore, by preallocating the necessary storage space up front, as is done in Listing 6.1, one can make the time it takes to populate (assign values to) an array or a matrix $O(n)$ relative to the size n of the array.

I performed an experiment in which I allocate arrays statically and dynamically. I compare the time it takes to preallocate an array and use a for loop to set each of its values to 1 to the time it takes to dynamically allocate and append the number 1 just as many times. I increase the length of this array from 10 numbers to $10,000$ floating point numbers.

I found that statically allocating and initializing arrays is consistently two to three orders of magnitude faster than dynamically allocating and initializing them. In both cases, allocation time is an increasing function of array size. The results of this experiment are shown in Figure B-5.

Executing Critical Sections in Other Runtimes

Finally, the most subtle yet powerful way to optimize performance in time at the statement level is to externalize individual statements into other execution runtimes. First, one uses the profiler (in MATLAB) to find statements that require a large percentage of the total run time, and at the same time appear to have a time performance that scales with the size of the input data. Such computationally intensive statements are known as critical sections. Next, one analyzes whether the computation is parallelizable, and whether changing the basis of the input data enables the computation to be done faster. Finally, one must choose the interface for externalization. One could use files as an interface to local programs, or the network as an interface to a collection of networked computers, or direct memory access as the interface to dynamically loaded binaries.

I used the MATLAB profiler to find a statement in the scalogram visualization listed in Section 5.2.2 that was making scalogram computation take on the order of one hour for a time record with $100,000$ samples. Convolution of complex signals was being done in the time domain ($O(n^2)$) in this statement. Convolution is faster in the frequency domain ($O(n \lg n)$), and convolution is easily parallelizable when the inputs are in the frequency domain, as convolution becomes elementwise multiplication. So, I decided to externalize this convolution to a C library that took advantage of Intel processor-specific optimizations in the IPP (Intel Processing Primitives) signal processing library. By externalizing this convolution, I was able to reduce the computation time for the scalogram visualization from one hour to around thirty seconds, given a time series $100,000$ samples long.

6.2 Preparation to Migrate to Free Development Platforms

Introduction of ETL processes, object orientation with classes, and separation of the view (plotting API), model (VIV event domain), and distinct service / controller layers

(MVC) made VIV Suite a modern application that takes advantage of large amounts of physical RAM. In particular, I demonstrated that the addition of OOP with classes and the separation of application function into MVC layers enabled the visualizations to be rendered many orders of magnitude faster than was previously possible.

Introducing OOP and the MVC design paradigm not only made the application modern from an architectural point of view, it also completed the first step of porting VIV Suite in its entirety to another platform. Currently, VIV Suite runs in proprietary (MATLAB) and obsolete (Fortran) runtimes. The cost of maintaining VIV Suite in the current platform is high, will rise as the number of MATLAB licenses required increases, and is expected to rise even more as support for Fortran programs becomes harder to find. Therefore, it is important that cost and deprecation risk be mitigated by planning for migration to another platform.

Walling off the view layer into one group of classes (the Plotting API in Section 5.1) makes it apparent exactly what plotting code needs to be implemented or adapted to in another language. Again, using object-oriented classes throughout the application makes porting VIV Suite to another language, such as C++ or Java, straightforward.

Chapter 7

Conclusion

To conclude, I will summarize my contributions to optimizing VIV visualization using VIV Suite, and suggest routes for future work.

7.1 Contributions

In this thesis, I composed a detailed description of the data domain of VIV events (Chapter 2). After analyzing the domain model, I gave an overview of the design of VIV Suite, a collection of VIV visualizations (Chapter 3). Next, I described the domain model of VIV events and VIV event collections in the context of VIV Suite (Chapter 4). I contributed efficient implementations of five time-domain, space-domain, and frequency domain visualizations of VIV, described in detail in Chapter 5. Subsequently, I demonstrated with empirical evidence (Chapter 6) that the modern application paradigms I integrated in VIV Suite, such as object-oriented programming (OOP), model-view-controller (MVC), and dynamic binary loading, contributed to an implementation many orders of magnitude more efficient than its legacy implementation.

7.2 Future Work

In the future, I can see a number of extensions to VIV Suite that range broadly from architectural changes to domain model changes to localized optimizations.

The largest and most important architectural change to VIV Suite is migration from MATLAB/Fortran to C++. The first step in this migration would be to mitigate execution risk by making sure all visualizations are exactly reproducible using a graphics API that has handles in C or C++. For example, one keeps the interface of the Plotting API but reimplement its back end to use another graphics library, such as GNU plot or OpenGL. Finally, automating VIVA simulation runs to evaluate large numbers of hydrodynamic databases is another step necessary to improve the current VIV model. Automating VIVA simulation runs would require more domain state to be represented in `VivEvents`.

VIV Suite's domain model is by no means complete; many hydrodynamic and structural properties of VIV events, which are particular to each experiment, are defined in scripts that generate VIVA simulator input files. As these properties are constant for each experimental configuration, they may as well be defined in the ETL `Iterator` for each configuration, and abstracted and represented as VIV event states in the `VivEvent` class. Examples of VIV events' structural properties not yet represented in the `VivEvent` abstraction, but required by the VIVA simulator include the following: tension, segment length, inner and outer riser radii, and segment strake configuration. Examples of hydrodynamic properties not yet represented in the `VivEvent` abstraction include Reynolds number, fluid density, and a time-variant representation of the event's flow profile.

Experimental strain and displacement RMS are the guidelines against which VIVA simulation results are evaluated in the VIVOS visualization. Thus, finding hydrodynamic databases that minimize the difference between experimental and simulated RMS is the goal of automating VIVA simulator and VIVOS runs. Currently, experimental RMS data are generated and stored as a low-fidelity, intermediate representation of a VIV event alongside the raw event data. Since generating RMS data from a VIV event is a computationally intensive process, one could write a C library and use dynamic binary loading to make RMS generation a real-time procedure, just as scalogram computation has become. Thus, RMS data can be generated on the fly, enabling the VIVOS visualization to depend solely on VIVA output and experimental configuration-specific iterators.

Appendix A

Tables

Table A.1: Execution environment for optimization strategy tests

Item	Specification
Processor Make	Intel
Processor Model	Xeon
Processor Die	2
Cores per die	4
Total cores	8
L2 Cache (per core)	256 KB
L3 Cache (per die)	8 MB
Memory	16 GB
Physical Store	Hardware RAID 5 3+1 1.67 TB

Table A.2: Classification of Currently Available Data Sets

Name	**Experiment Type**	**Diameter(m)**	**Length(m)**	**Max Flow(m/s)**
Chaplin	Controlled	0.028	13.1208	0.9512
DEN	Field	1.3081	1738.9	1.51
DENNEW	Field	1.3081	1738.9	1.37
Miami	Field	0.0363	152.5	2.6992
NDP10	Controlled	0.02	9.63	2.38
NDP38	Controlled	0.028	38	2.4
Schiehallion	Field	1.207	378.8	1.24

Appendix B

Figures

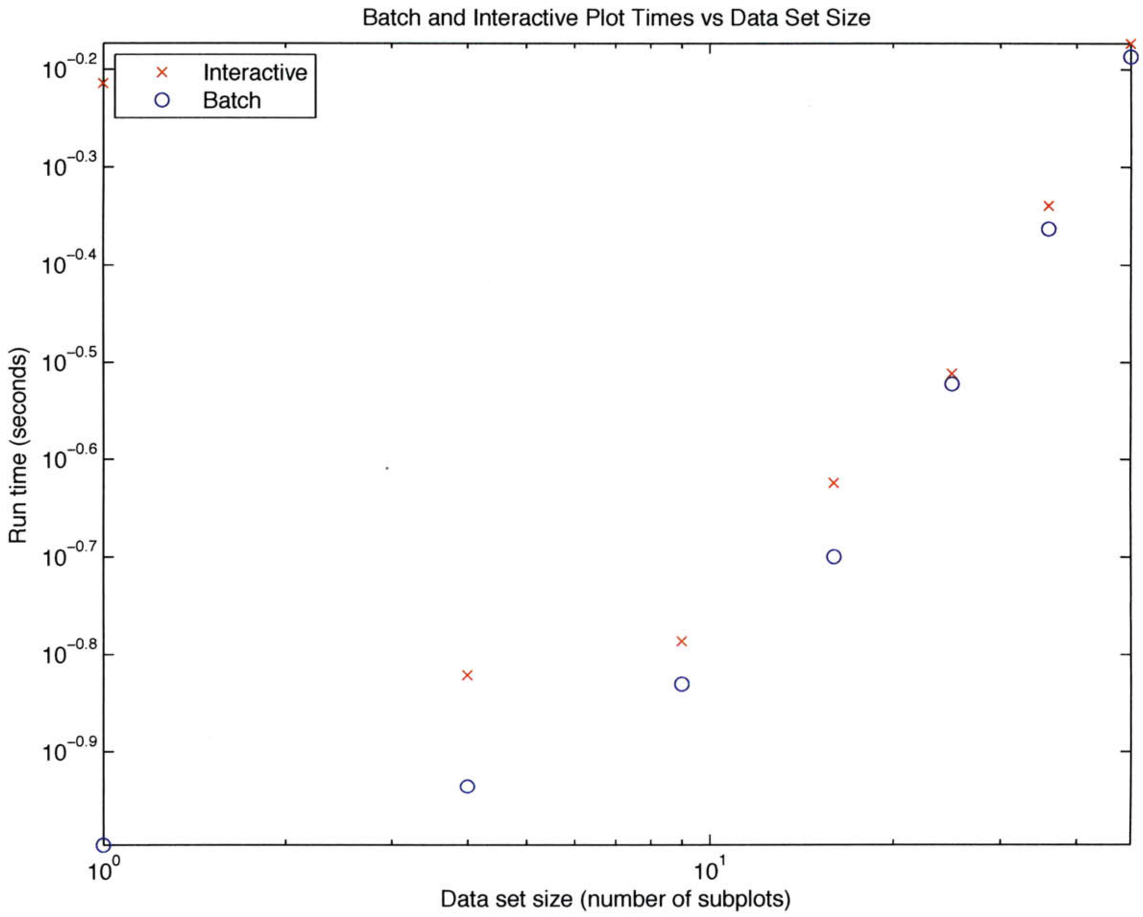

Figure B-1: A comparison of computation and render time for MATLAB subplots generated interactively and in batch.

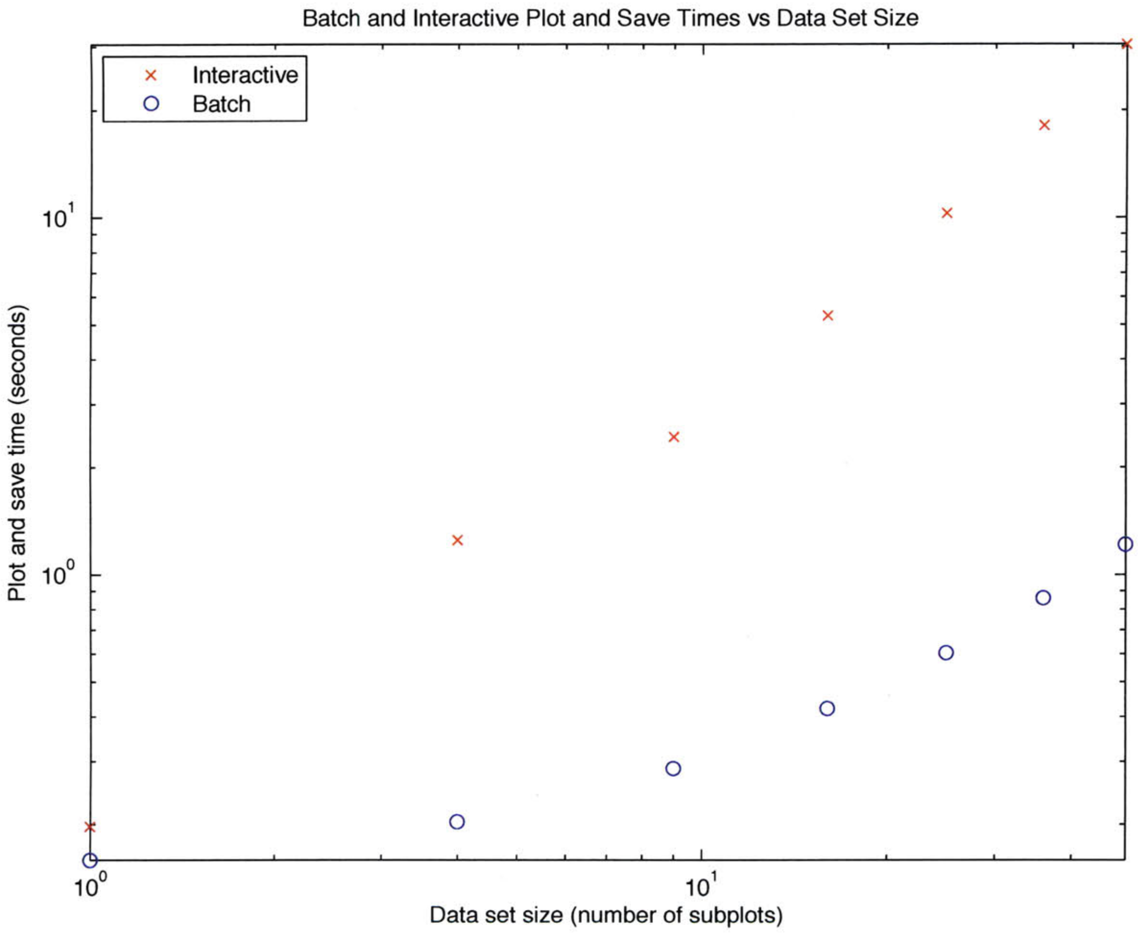

Figure B-2: A comparison of computation, render, and save time for MATLAB subplots generated and saved interactively and in batch.

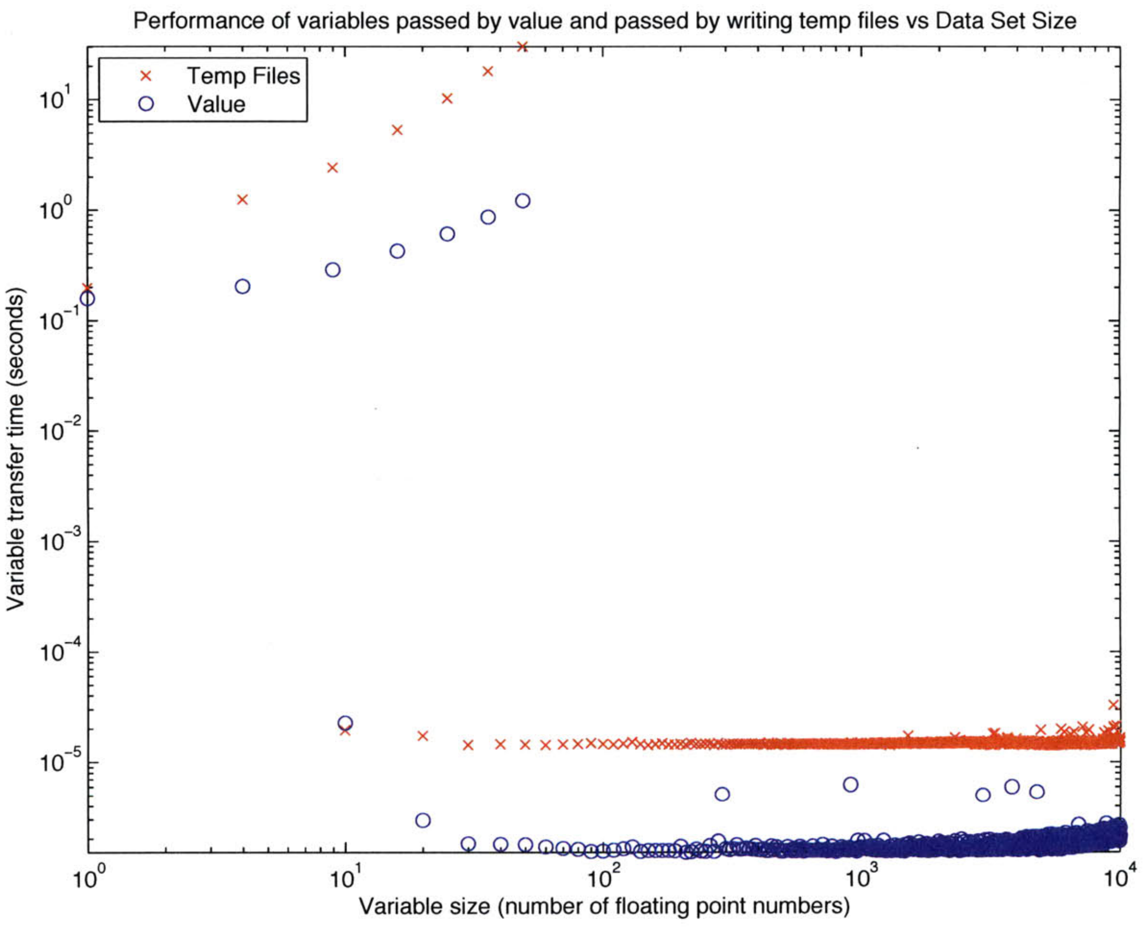

Figure B-3: A comparison of frame switch times when passing variables by value, and when passing by using temporary files.

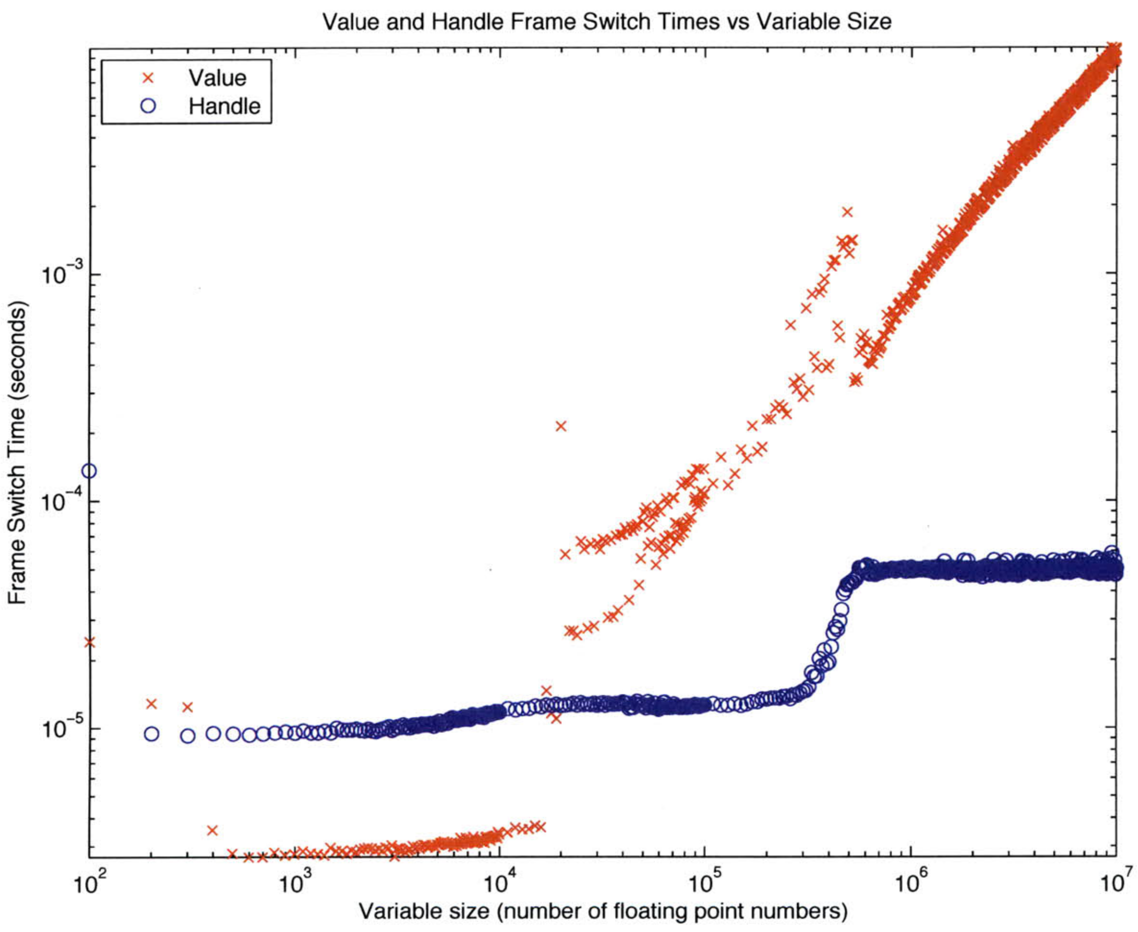

Figure B-4: A comparison of frame switch times when passing variables by value, and when passing by using handle class instances.

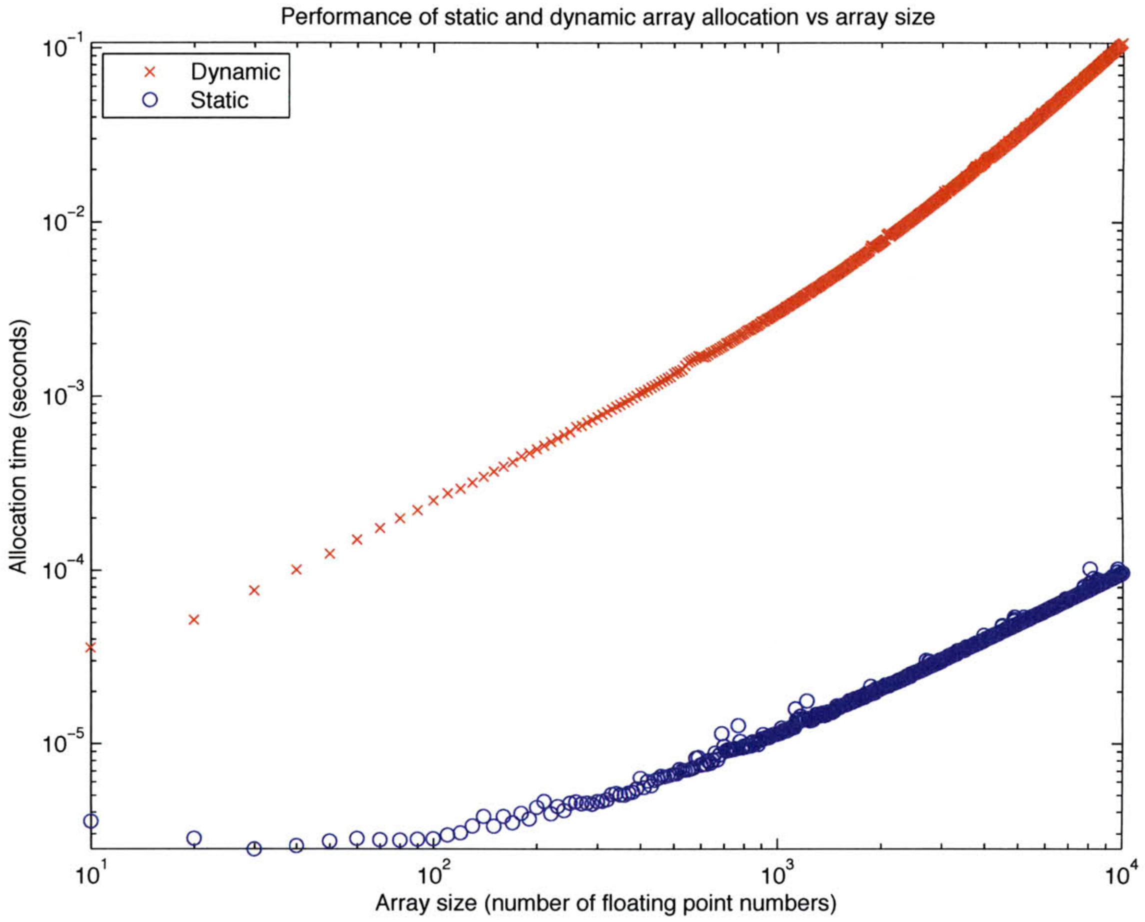

Figure B-5: A comparison of static and dynamic array allocation times versus array size

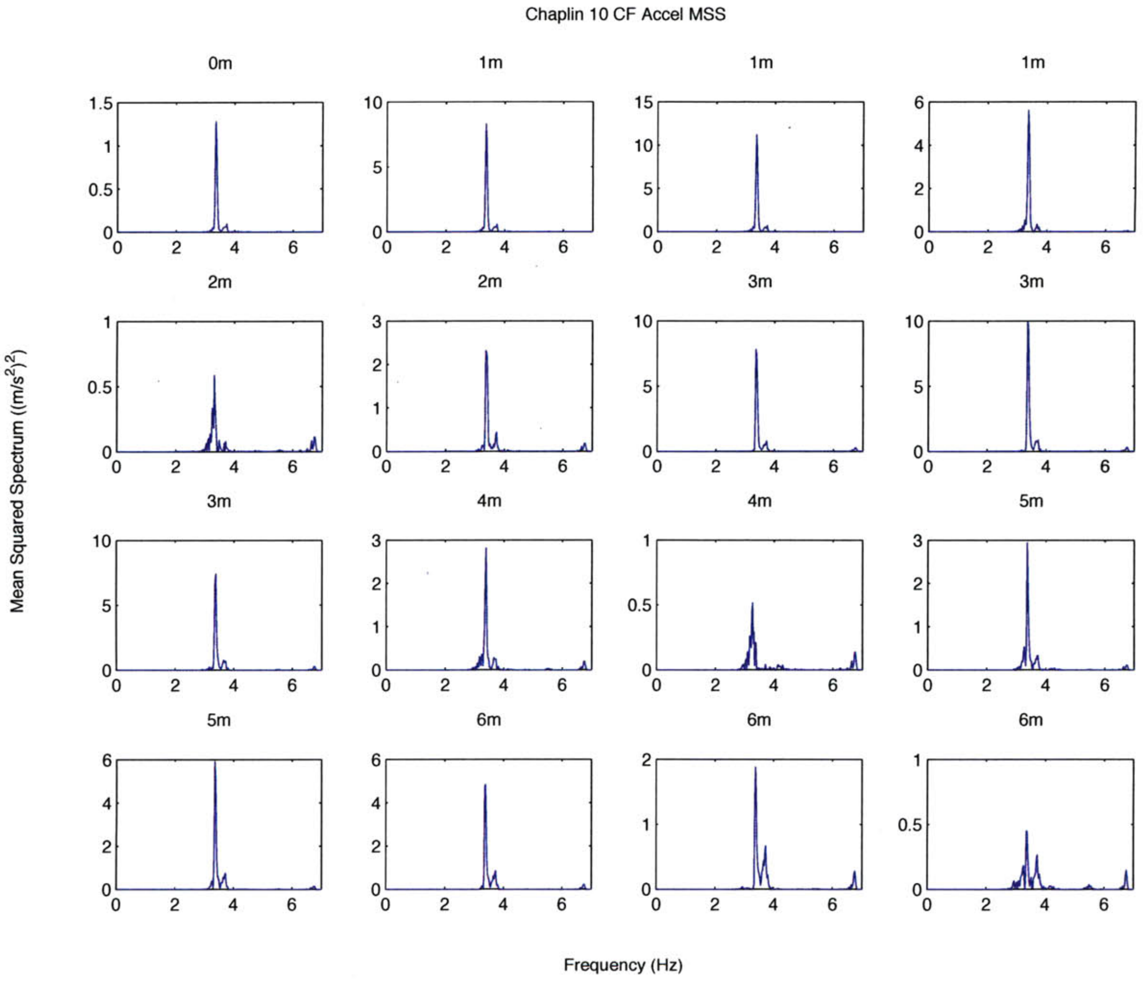

Figure B-6: The mean-squared spectrum of the acceleration signals measured at each of the accelerometers along the span of the riser in Chaplin event number 10. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

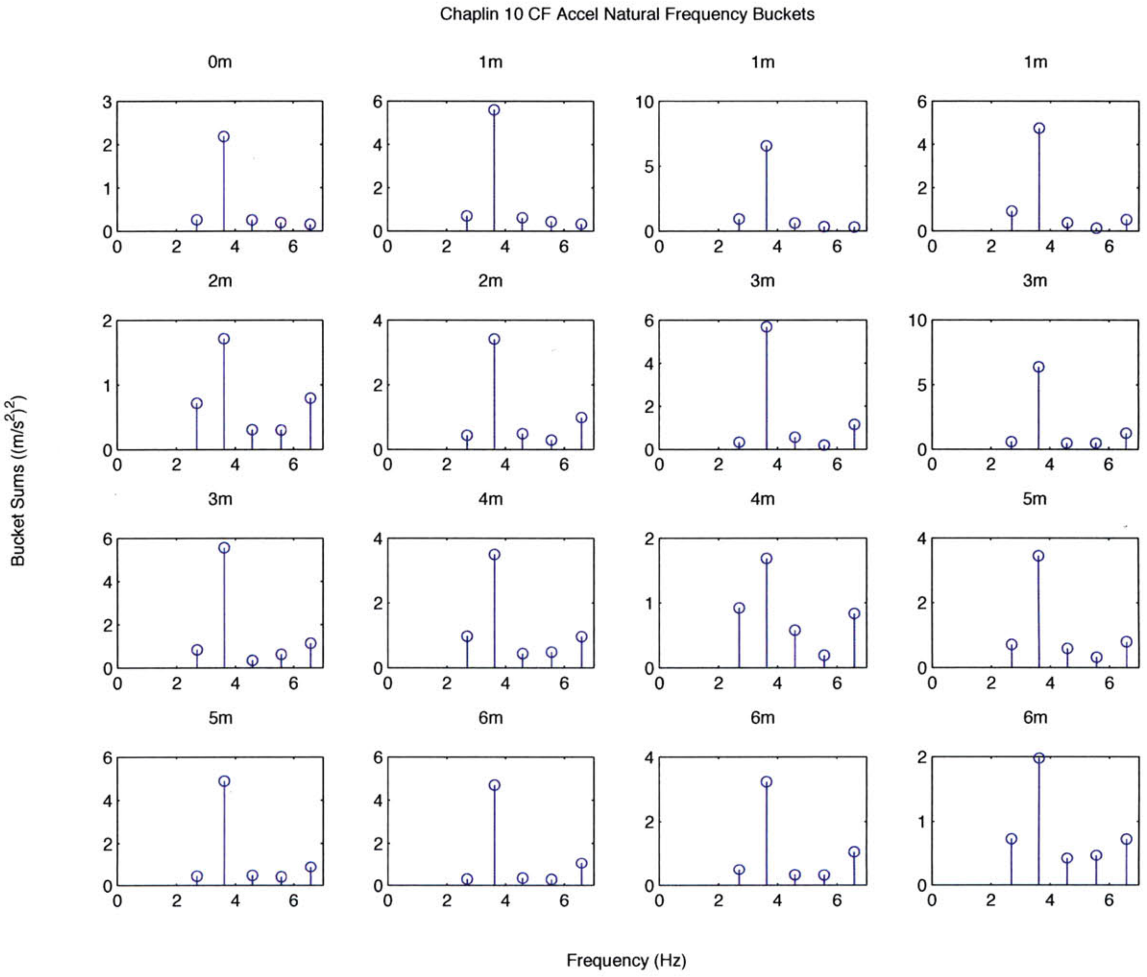

Figure B-7: The natural frequency content of the acceleration signals along the span of the riser in Chaplin event numer 10, as computed from its mean squared spectrum. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

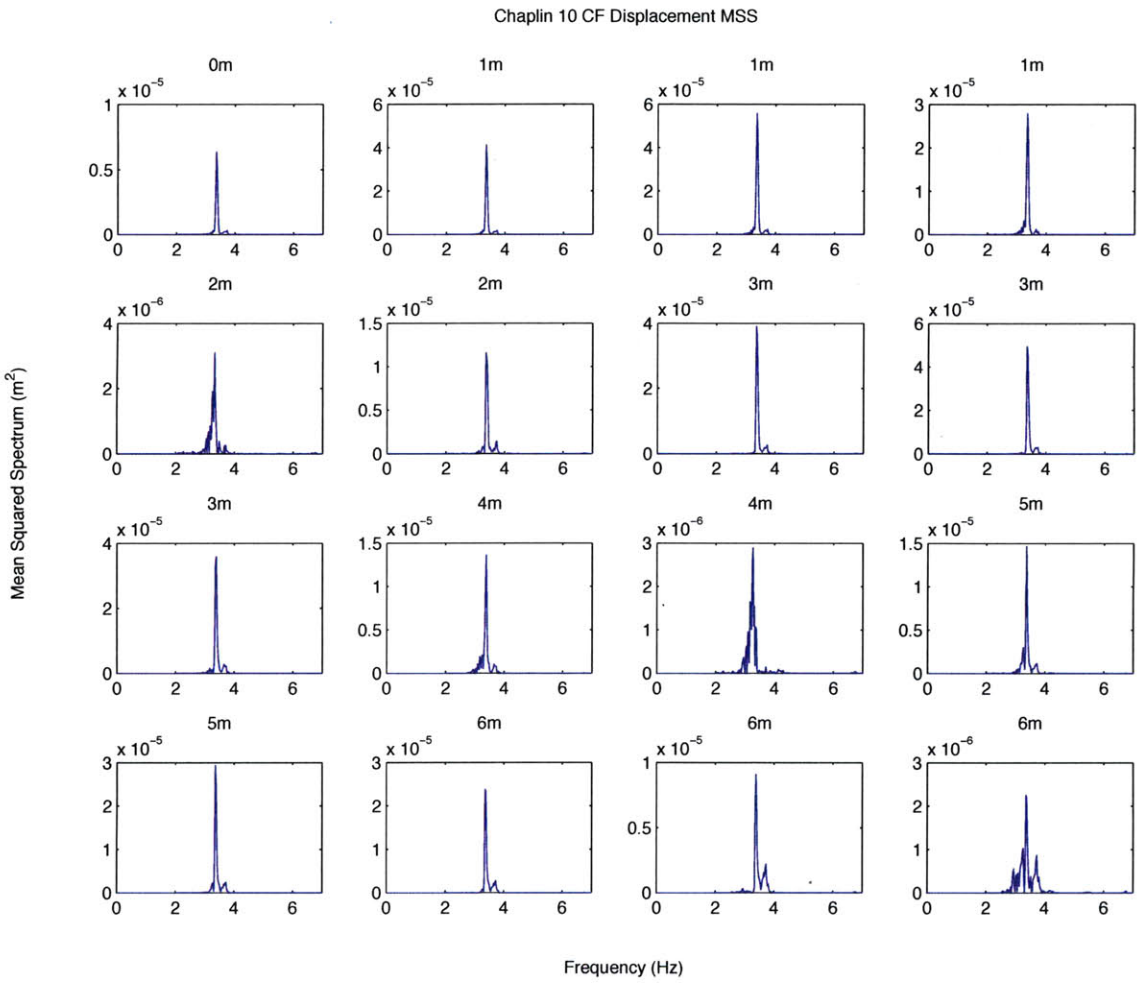

Figure B-8: The natural frequency content of the displacement corresponding to each accelerometer in Chaplin event number 10. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

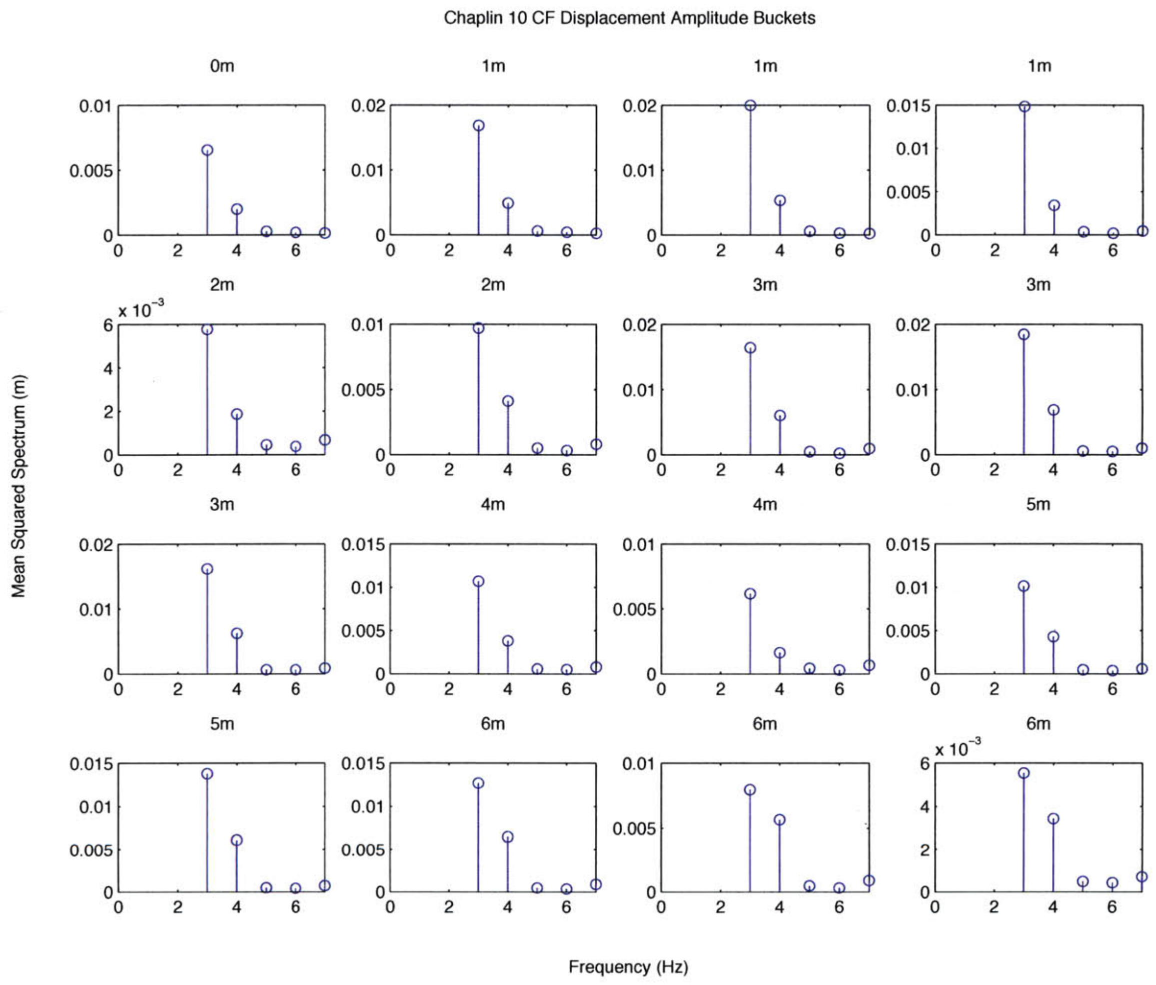

Figure B-9: The natural frequency content of the displacement amplitude, corresponding to each accelerometer in Chaplin event number 10. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

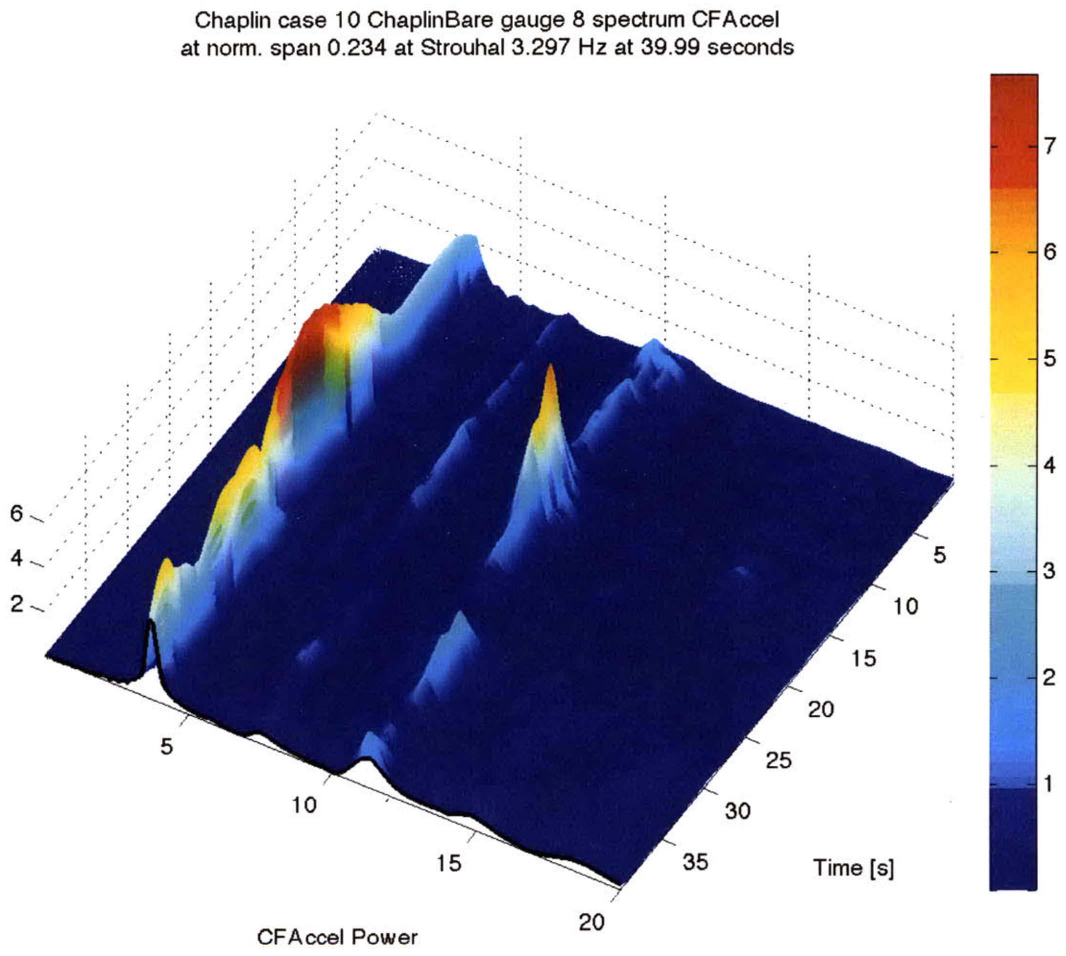

Figure B-10: A scalogram of the CF acceleration measured at the eighth accelerometer in Chaplin event 10. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

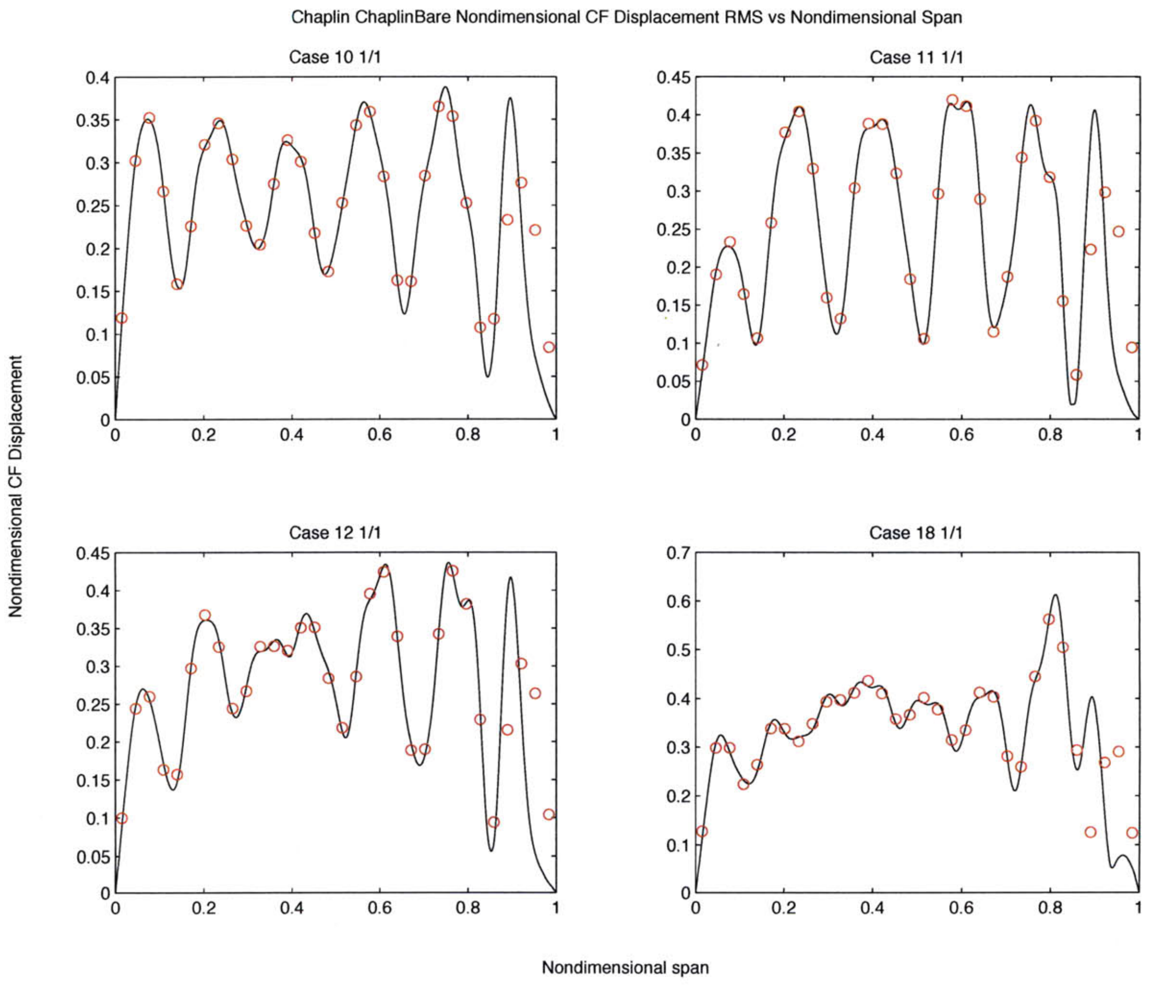

Figure B-11: Summary plots of reconstructed displacement RMS with experimental displacement RMS values superimposed thereupon for Chaplin events numbers 10, 11, 12, and 18. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is 0.54, 0.60, 0.65, and $0.95m/s$, respectively.

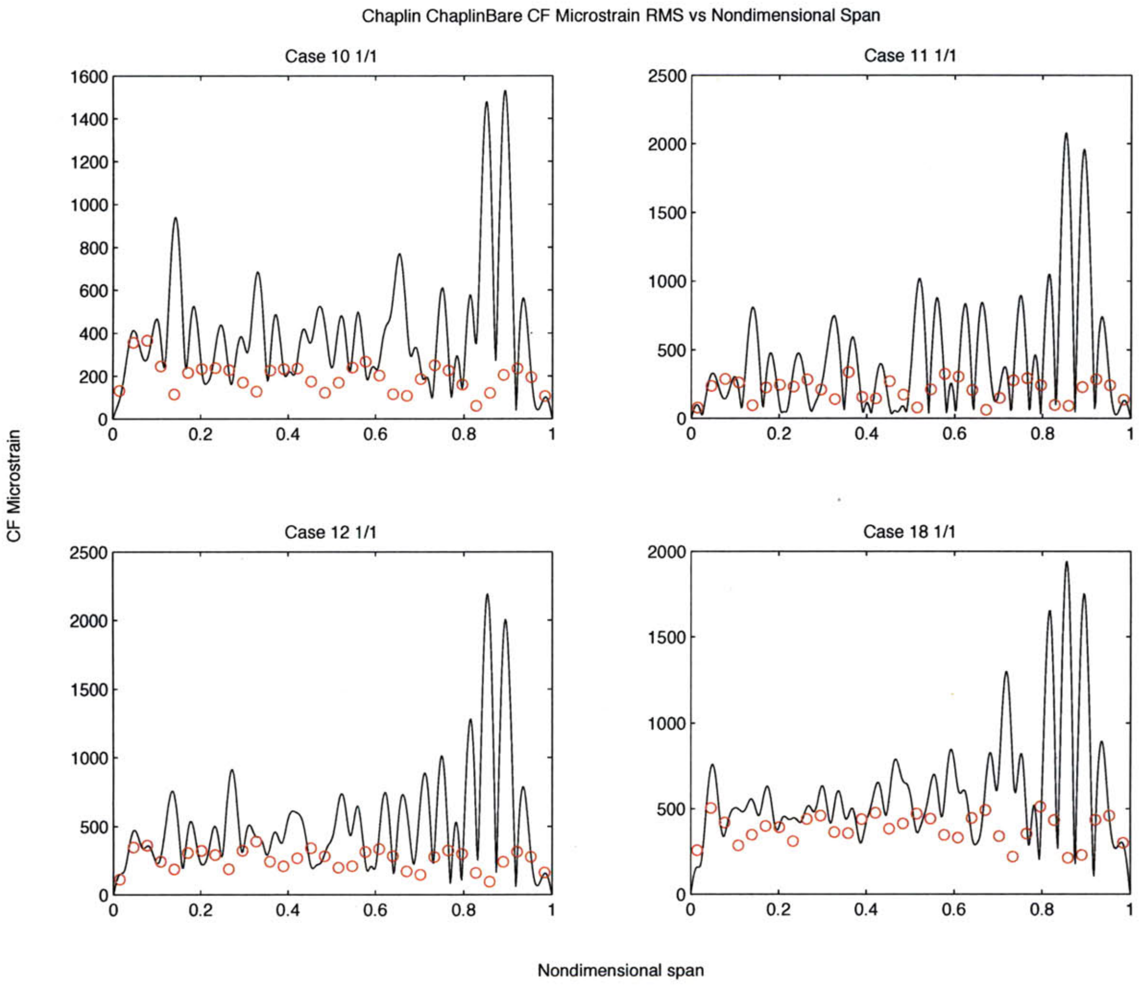

Figure B-12: Summary plots of reconstructed strain RMS with experimental strain RMS values superimposed thereupon for Chaplin event numbers 10, 11, 12, and 18. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is 0.54, 0.60, 0.65, $0.95m/s$, respectively.

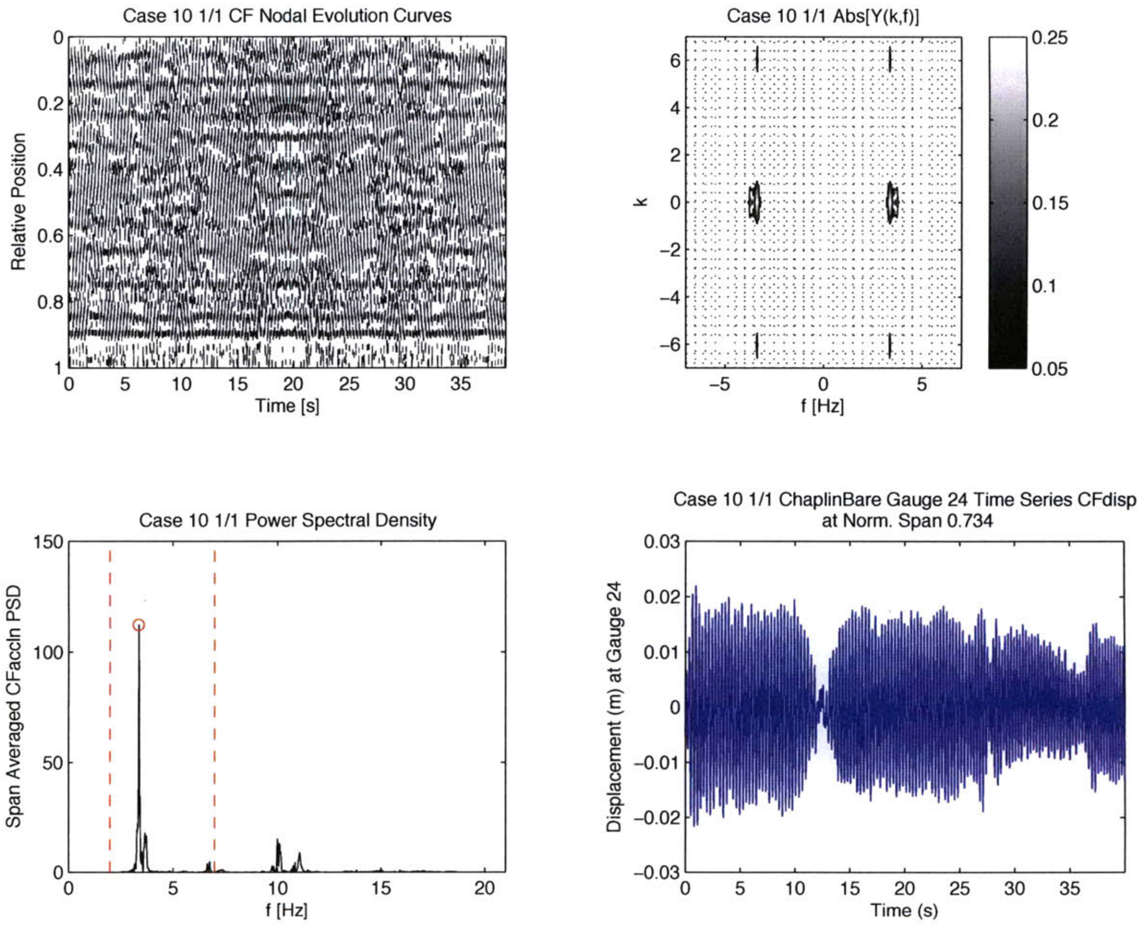

Figure B-13: Visualization of the reconstruction of Chaplin event number 10. The top-left plot is the nodal crests plot that differentiates between standing and traveling waves in a VIV event. The top-right plot is the 2-D Fourier transform of the 500 reconstructed acceleration signals uniformly distributed along the span of the riser. The bottom-left plot is the span-averaged PSD of CF acceleration. The bottom-right plot shows the CF displacement time series at the 75% span point along the riser. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.54m/s$.

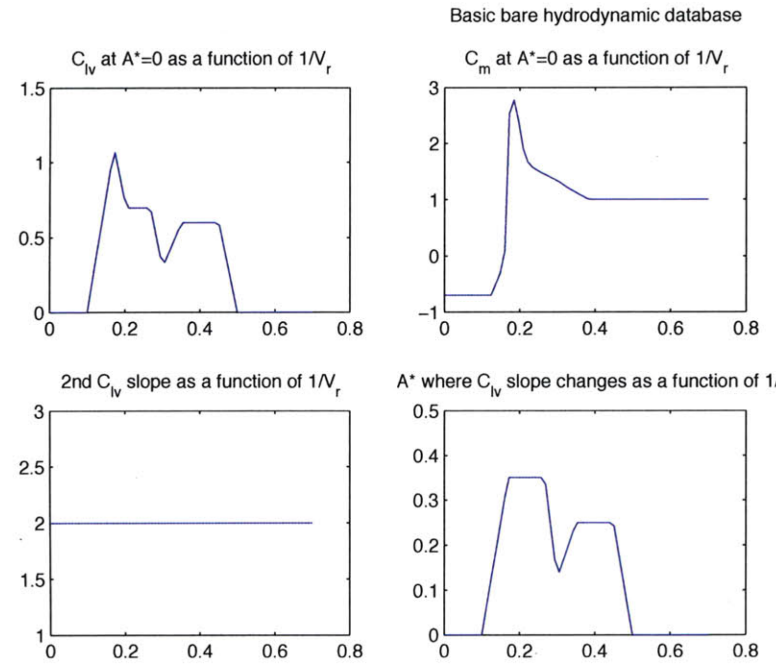

Figure B-14: The `basic_bare` hydrodynamic database. These five plots, from left to right and top to bottom, are all functions of reduced (nondimensional) frequency. The first plot displays the lift coefficient in phase with velocity. The second plot shows the added-mass coefficient. The third plot represents the first slope of the lift coefficient in phase with velocity. The fourth plot is the second slop of the lift coefficient in phase with velocity. Finally, the fifth plot displays nondimensional amplitude $A^* = A/D$, where the slope changes from the first to the second slope.

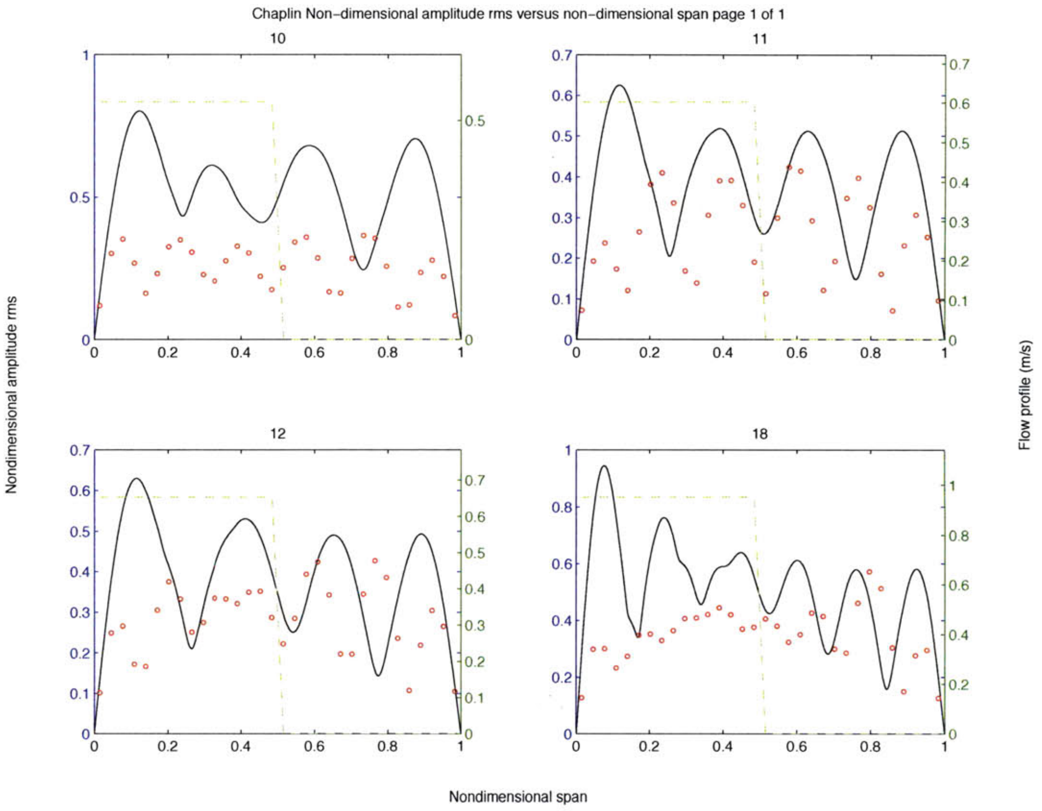

Figure B-15: Summary plots of simulated displacement amplitude RMS with experimental displacement amplitude RMS values superimposed thereupon for Chaplin event numbers 10, 11, 12, and 18. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocities are 0.54, 0.60, 0.65, and $0.95m/s$ respectively.

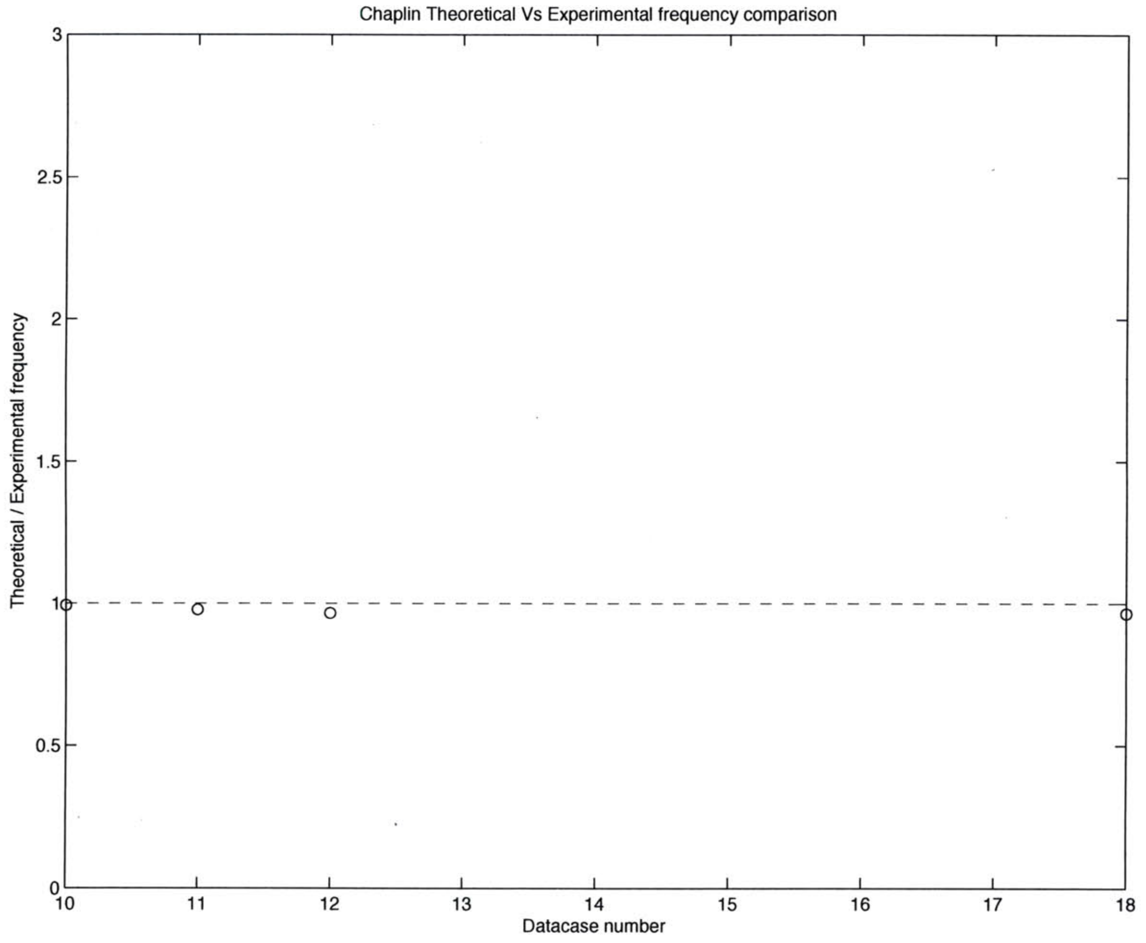

Figure B-16: Plot of the ratios of simulated harmonic frequency to experimental harmonic frequency for the Chaplin event numbers 10, 11, 12, and 18. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocities are 0.54, 0.60, 0.65, and $0.95m/s$ respectively.

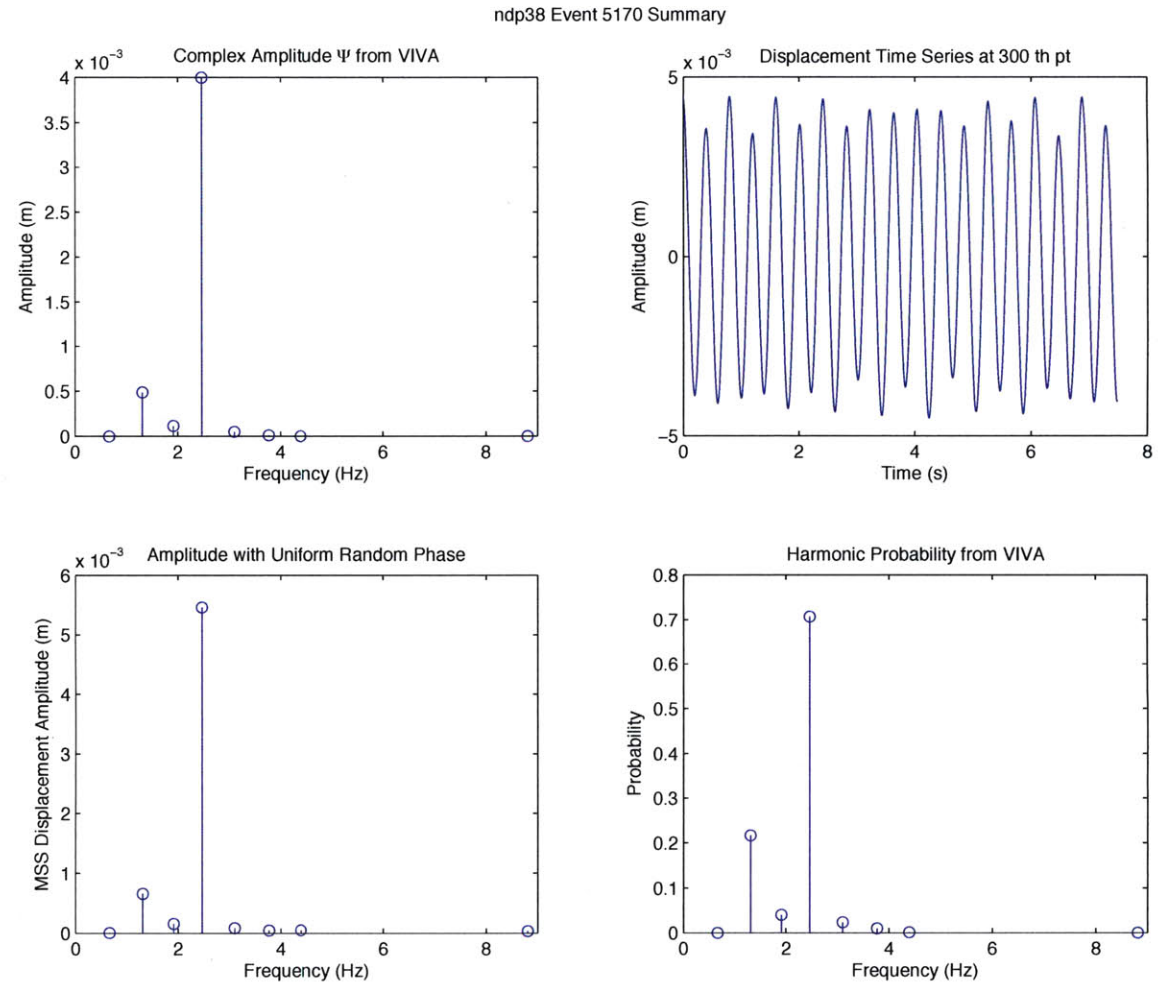

Figure B-17: NDP Sheared Straked 50% event number 5170 harmonic frequency and amplitude summary plot. From left to right and top to bottom, the first plot is a plot of the VIVA prediction of complex amplitude of CF displacement harmonics at a particular point along the span of the riser. The next plot is the time-domain representation of CF displacement at the same point on the riser. The third plot is shows the VIVA harmonics and their complex amplitudes in the frequency domain with random phase ϕ added. The random phase has a uniform distribution over the interval $[0, 2\pi]$. Finally, the last plot shows the VIVA prediction of the probability that each harmonic will exhibit itself in a marine riser, given the same experimental conditions. The length of the riser is $38m$, tension at the top of the riser is $5000N$, and the flow velocity is $0.9m/s$.

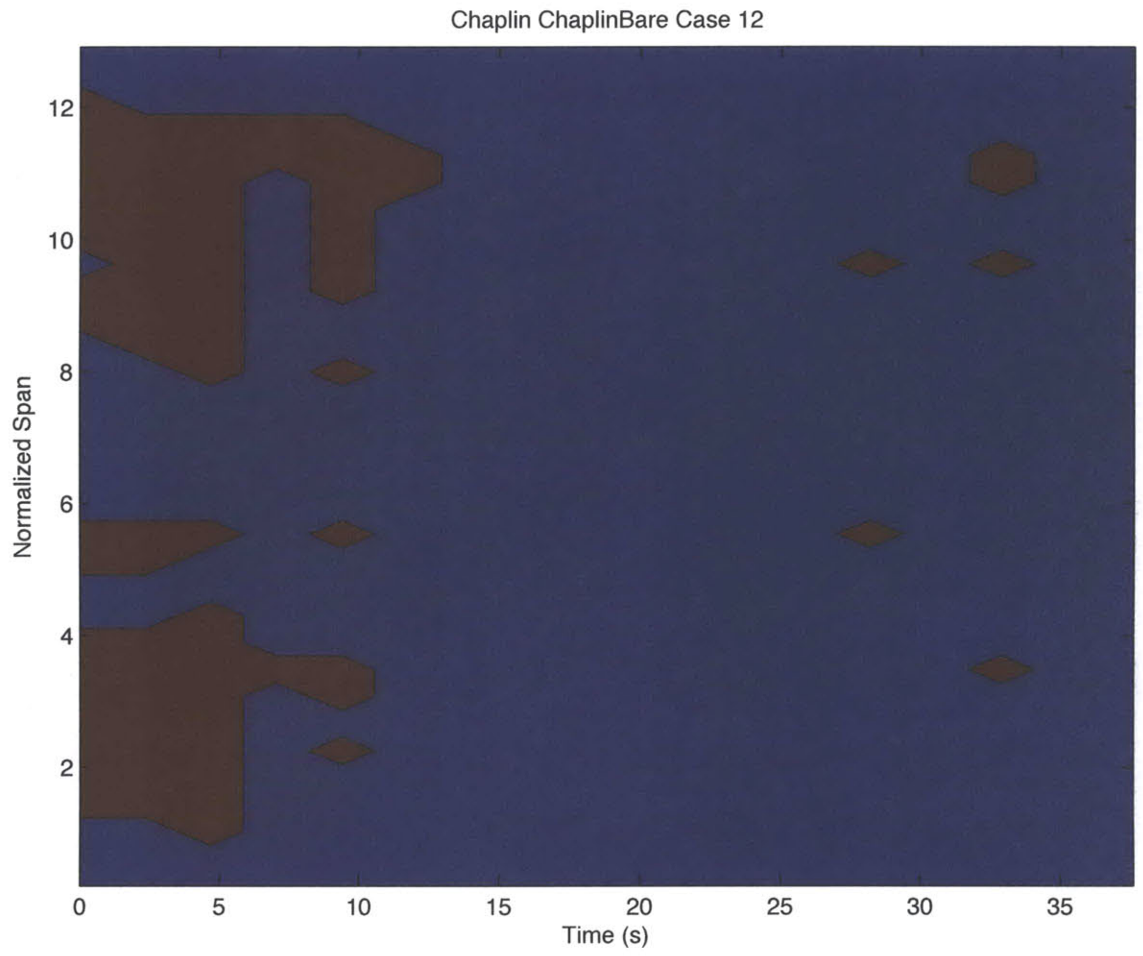

Figure B-18: The response analysis visualization for Chaplin event number 12. The blue regions indicate steady-state response, while the red regions indicate chaotic response. The length of the riser is $13.12m$, tension at the top of the riser is $800N$, and the flow velocity is $0.65m/s$.

Bibliography

[1] Halvor Braaten, Henning; Lie. NDP riser high mode VIV tests main report. Technical report, Norwegian Marine Technology Research Institute., 2004.

[2] J.R. Chaplin, P.W. Bearman, Y. Cheng, E. Fontaine, J.M.R. Graham, K. Herfjord, F.J. Huera Huarte, M. Isherwood, K. Lambrakos, C.M. Larsen, J.R. Meneghini, G. Moe, R.J. Pattenden, M.S. Triantafyllou, and R.H.J. Willden. Blind predictions of laboratory measurements of vortex-induced vibrations of a tension riser. *Journal of Fluids and Structures*, 21(1):25 – 40, 2005. Fluid-Structure and Flow-Acoustic Interactions involving Bluff Bodies.

[3] Filippos Chasparis. Vortex-induced motions of marine risers: Straked force database extraction and transient response analysis. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, June 2009.

[4] Richard Harris. BP's Own Numbers Prove Spill Greater Than Estimate. *National Public Radio*, May 2010.

[5] Clifford Krauss and Elisabeth Rosenthal. The Price and Who Pays: Updates From the Gulf. *New York Times*, page A18, May 13 2010.

[6] 2H Offshore Engineering Ltd. Riser VIV response parameter review and visualization. Technical report, 2H Offshore Engineering Ltd, 2003.

[7] James C. McKinley Jr. and Campbell Robertson. Oil Is Fouling Wetlands, Official Says. *New York Times*, page A20, May 20 2010.

[8] H. Mukundan. *Vortex-induced vibrations of marine risers: motion and force reconstruction from field and experimental data.* PhD thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, June 2008.

[9] T. Sarpkaya. A critical review of the intrinsic nature of vortex-induced vibrations. *Journal of Fluids and Structures*, 19(4):389 – 447, 2004.

[10] G. Triantafyllou. Vortex induced vibrations of long cylindrical structures. In *Proceedings of the ASME Summer Meeting*, volume 50, pages 1–8, Washington, DC, 1998. American Society of Mechanical Engineers.

[11] M. Triantafyllou, G. Triantafyllou, Y. Tein, and B. Ambrose. Pragmatic riser VIV analysis. In *Offshore Technology Conference*, Houston, TX, May 1999.

[12] Michael S. Triantafyllou. The dynamics of taut inclined cables. *Journal of Mechanics and Applied Mathematics*, 37:421–440, 1984.

[13] Michael S. Triantafyllou. *VIVA: Programs for Calculating Riser Vortex Induced Oscillations and Fatigue Life.* Massachusetts Institute of Technology, Testing Tank Facility, Cambridge, MA, 2006.

[14] J. K. Vandiver, H. Marcollo, S. Fantone, V. Jaiswal, V. Jhingran, and S. Swithenbank. DeepStar 7402 Miami 2004 Riser Report. Technical report, Massachusetts Institute of Technology, Department of Ocean Engineering, 2004.