

Flute acoustics: measurement, modelling and design

by

Paul A. Dickens
BSc(Hons), *Syd*

In fulfillment
of the requirements for the degree
Doctor of Philosophy

School of Physics
University of New South Wales
November 2007

Copyright © 2007 by Paul A. Dickens

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only). I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.

To Renée

Acknowledgements

This thesis would not be possible without the collaboration and assistance of many people.

Firstly, many thanks to my supervisor Joe Wolfe, for well-timed encouragement, motivation and understanding throughout my candidature, especially regarding the varied demands of having a young family. Joe's commitment to integrity in research is a powerful example, supported by his willing comradeship through many laborious late-nights experiments. Joe (with Andrew Botros) also conducted the flute tuning measurements of Chapter 7 in my absence. Thanks to my co-supervisor John Smith for his many helpful insights and for sharing his ideas as the project progressed. Joe and John collaborated with me on the writing of papers that published the material reported in Chapters 3 and 5. Their comments on early drafts of the thesis are much appreciated.

Thanks to John Tann for invaluable and varied technical assistance, for a positive, 'can-do' approach to the many problems I brought to him, and for the many illuminating discussions resulting from my sometimes erratic and ill-formed questions.

Much of the software code for the flute model is derived from the excellent work of Andrew Botros. Andrew gave up his own time to help with issues of software design and his generosity in this is appreciated.

Thanks to Terry McGee and Mark O'Conner for graciously imparting their practical knowledge in the art and science of flute making, to Terry for taking me on as apprentice for a day and for being a guinea-pig in the development of the flute design software. Terry's insights about classical flute tuning form part of the discussion in Chapter 9. Thanks to the Powerhouse Museum, Sydney and the museum's curator of musical instruments, Michael Lee, for access to their collection of instruments as well as financial support of the project. The collaboration with Terry and Michael was made possible through an Australian Postgraduate Award (Industry) granted by the Australian Research Council.

Flutists Jane Cavanagh, Cécile van der Burgh, Caoimhe McMillan and clarinettist Catherine Young added a much-needed human dimension to woodwind modelling. I am grateful for the way in which each one cheerfully cooperated with uncomfortable and repetitive experiments.

My thanks to fellow students in the Acoustics lab, particularly Claudia Fritz and Ra Inta, for helping to keep my spirits up and for many fascinating discussions (some of which were about acoustics).

Ken Jackson and Pritpal Baweja in the Physics workshop turned my hasty sketches into beautiful creations in brass and aluminium—their ingenuity and craftsmanship is appreciated.

I am supported by a diverse community of people with whom I am linked by our common Christian faith. I particularly wish to thank Ian, Gina and Amanda for their hopeful, frequent and sincere encouragement in this endeavour, and for helping provide perspective when the trees got in the way of the forest.

I am grateful to my parents, Ross and Valerie, and to my siblings and siblings-in-law for the gift of constant love, which frequently took the form of weeknight dinners.

Finally, my humble thanks to my wife Renée. Her support and trust through my candidature has been a great blessing and source of strength. And to my son Oliver, for his one-year-old's sense of fun.

Abstract

A well-made flute is always a compromise and the job of flute makers is to achieve a musically and aesthetically satisfying compromise; a task that involves much trial-and-error. The practical aim of this thesis is to develop a mathematical model of the flute and a computer program that assists in the flute design process.

Many musical qualities of a woodwind instrument may be calculated from the acoustic impedance spectrum of the instrument. A technique for fast and accurate measurement of this quantity is developed. The technique is based on the multiple-microphone technique, and uses resonance-free impedance loads to calibrate the system and spectral shaping to improve the precision at impedance extrema. The impedance spectra of the flute and clarinet are measured over a wide range of fingerings, yielding a comprehensive and accurate database. The impedance properties of single finger holes are measured using a related technique, and fit-formulae are derived for the length corrections of closed finger holes for a typical range of hole sizes and lengths.

The bore surface of wooden instruments can change over time with playing and this can affect the acoustic impedance, and therefore the playing quality. Such changes in acoustic impedance are explored using wooden test pipes. To account for the effect of a typical player on flute tuning, an empirical correction is determined from the measured tuning of both modern and classical flutes as played by several professional and semi-professional players. By combining the measured impedance database with the player effects and various results in the literature a mathematical model of the input impedance of flutes is developed and implemented in command-line programs written in the software language C.

A user-friendly graphical interface is created using the flute impedance model for the purposes of flute acoustical design and analysis. The program calculates the tuning and other acoustical properties for any given geometry. The program is applied to a modern flute and a classical flute. The capabilities and limitations of the software are thereby illustrated and possible contributions of the program to contemporary flute design are explored.

Contents

Dedication	iii
Acknowledgements	v
Abstract	vii
 CHAPTERS	
I Introduction	1
1.1 Introduction to flute acoustics	1
1.2 The flute maker, the curator and the software engineer	2
1.3 ‘FluteCAD’: taking the guesswork out of flute design	3
1.4 How flutes work	5
1.5 Acoustic impedance: predicting the playing qualities of a flute	6
1.6 Tuning, tradition and the ‘standard flutist’	6
1.7 Software implementation: goals and limitations	8
1.8 Guide to the thesis	8
II Theory and literature review	11
2.1 Flutes and flute making	11
2.2 Acoustics of woodwind instruments	14
2.3 Computer models	29
III Measuring acoustic impedance	31
3.1 Introduction	31
3.2 Review of measurement techniques	32
3.3 Theory of acoustic impedance measurements	32
3.4 Calibration of impedance heads	36
3.5 Errors	40
3.6 Optimisation of the output signal	45
3.7 Materials and methods	45
3.8 Results and discussion	47
IV Finger hole impedance spectra and length corrections	53
4.1 Introduction	53
4.2 Materials and methods	55
4.3 Results and discussion	59

4.4	Conclusions and further directions	66
V	Impedance spectra of the flute and clarinet	69
5.1	Introduction	69
5.2	Materials and methods	69
5.3	Results and discussion	72
VI	Material and surface effects	97
6.1	Introduction	97
6.2	Materials and methods	97
6.3	Results and discussion	98
6.4	Further investigations	102
VII	The embouchure hole and player corrections	103
7.1	Player impedance corrections	103
7.2	Modern flute tuning	104
7.3	Correction factor	106
7.4	Application to the classical flute	107
7.5	Conclusions and further directions	108
VIII	Software implementation	111
8.1	The impedance model	111
8.2	The user interface	119
IX	Applications and further directions	127
9.1	Adding a new hole to the modern flute	127
9.2	Eight-key flute by Rudall & Rose	132
9.3	Conclusions and further directions	138
Appendix A — Impedance spectra	141	
Appendix B — Program listings	195	
Appendix C — Quantifying music	337	
References	339	

Chapter I

Introduction

The world will never starve for want of wonders; but only for want of wonder.

G. K. Chesterton

1.1 INTRODUCTION TO FLUTE ACOUSTICS

Music and musical instruments have long fascinated physicists and philosophers. The simple numerical basis of musical harmonies is a striking example of the concordance between mathematics and aesthetics, and for many philosophers the mathematical theory of harmony has served as a metaphor for the quantitative understanding of the universe. For Pythagoras (c. 6th century BC) musical harmonies were the key to understanding the cosmos. The Pythagoreans likened planetary orbits to huge strings on which the universe itself made a kind of music—the ‘Harmony of the Spheres’. The orbits of each planet were related to those of its neighbours by a simple musical interval. This concept, although superseded by modern cosmology, captured the imagination of poets and scientists alike. The celestial music featured in the work of Shakespeare, Milton and Donne:

The spheres have music, but they have no tongue,
Their harmony is rather danced than sung ...

(Donne 1971, p. 333) and inspired Donne’s contemporary Kepler to devise his famous laws of planetary motion (Koestler 1959). Musical analogies also feature in Bohr’s model of the atom which led to modern quantum physics.

Quite apart from the influence of music in physics and philosophy, many scientists have been interested in music and musical instruments in their own right. In the 19th century Helmholtz and Rayleigh laid the foundations of musical acoustics and psychoacoustics. Many problems in musical acoustics are surprisingly complex and involve the interplay of physics, culture and psychology. Some musical instruments are a direct creative development of a scientific discovery—one thinks of the electric guitar and Faraday’s law—and instrument makers have historically made active use of scientific and technological developments.

Most musical instruments are made with many different (and sometimes conflicting) goals in mind. Instruments are usually required to play in tune over a wide range of pitches with reasonable timbral uniformity over the range (although in some musical traditions notes with different timbre are used for musical effect). In the case of the flute and other woodwind instruments the several dozen notes playable on the instrument are made when the player covers various holes in the instrument or operates various keys and the design of such instruments is constrained by the necessity for a relatively simple, responsive fingering mechanism. Inevitably, single holes on the instrument contribute to the production of more than one note,

and the position and size of such holes is a compromise. Existing instruments are therefore not perfect and even the idea of perfection in this context is contestable. Instruments usually change gradually in response to musical tastes and performance demands. There are some notable exceptions—such as Boehm’s development of the modern flute and the invention of the saxophone by Adolph Sax in the 19th century. Instrument makers vary in their approach to instrument design, some guided primarily by the work of earlier makers and simple rules-of-thumb, while others are more rigorous and acoustically informed. In most cases, however, an element of trial-and-error is inevitably involved.

In the light of this, one of the aims of music acoustics is to help makers: to better understand the physics of musical instruments and to use this knowledge to make better instruments, or even just to make good instruments more quickly or with less cost. Such is the goal of this thesis. The task is subtle and requires a high degree of precision, since the human ear is sensitive to minute changes in pitch and timbre, and any physical model needs to match that sensitivity. There are also many constraints on the design of musical instruments (determined by such things as the size of a player’s hands) and new designs need to fit within these constraints.

1.2 THE FLUTE MAKER, THE CURATOR AND THE SOFTWARE ENGINEER

This project presented the opportunity to bring together three diverse fields, experts in which might not typically find themselves in collaboration.

Terry McGee (shown at his mill in Figure 1.1) is an Australian maker of wooden flutes in the classical style. His flutes are shipped worldwide, particularly to players of Irish traditional music, of which he is one. Terry’s designs are based on popular historical instruments from the classical period (late 18th to early 19th centuries), but he regularly modifies his designs to improve intonation and better to suit customers’ requirements. Terry’s website <<http://www.mcgee-flutes.com/>>^{*} has extensive information on the history of flute making, his designs and innovations, and flute performance.

The Powerhouse Museum in Sydney houses a large collection of musical instruments. The museum has an extensive range of early flutes, many of which are made of wood and are now unplayable due to cracking or other mechanical defects. The museum’s curator of musical instruments, Michael Lee, is interested in the playing characteristics and intonation of these old instruments. This might potentially be used to trace the history of flute development, as well as changes in tuning and temperament over the last few hundred years.

The Virtual Flute is a web service built and maintained by software engineer Andrew Botros that uses a semi-empirical model of the impedance of the modern flute, based on measurements performed at the Music Acoustics Laboratory at the University of New South Wales. The web service calculates the impedance of the modern flute for any of its 39 744 possible acoustic configurations and provides players with tuning, timbre and playability predictions for these fingerings (Botros et al. 2006). The web service has many and diverse applications among players and composers such as finding new fingering combinations for difficult-to-play phrases,

^{*}A copy of this website archived by the National Library of Australia is available at <<http://nla.gov.au/nla.arc-24785>>.



Figure 1.1: Australian flute maker Terry McGee at work.

and finding fingerings for microtonics and multiphonics. The front page of The Virtual Flute is shown in Figure 1.2. The Virtual Flute currently resides at <<http://www.phys.unsw.edu.au/music/flute/virtual/>>.

The success of The Virtual Flute has led to requests for similar services for other instruments (such as the clarinet, oboe and bassoon) and has also suggested development in a parallel direction. Could we make a software model for the flute that would be accurate for arbitrary geometries, and able to model both the classical and modern flute and any designs in between? Makers such as Terry could use such a program to design new flutes, and museum curators such as Michael could examine old instruments with the program. Indeed, such is their interest that this project is co-sponsored by Terry McGee and the Powerhouse Museum.

1.3 ‘FluteCAD’: TAKING THE GUESSWORK OUT OF FLUTE DESIGN

The practical aim of this project is to make a versatile, accurate but user-friendly computer program to provide flute makers with useful predictions about the musical properties of a proposed instrument with a given geometrical design. Flute makers want answers to questions such as ‘Will this instrument be in tune?’ and ‘How difficult will this note be to play?’ If many of these questions can be answered before the production of a prototype, the time-to-market of new designs will be reduced markedly.

The program ‘FluteCAD’ needs to be accurate enough for music (a semitone corresponds to a 6% difference in frequency, and players are sometimes sensitive to differences of 5 or 10 cents, i.e. 0.3 or 0.6%), fast enough to calculate the pitch and playing qualities of a flute for

the virtual flute

Choose from any of the three tools below...

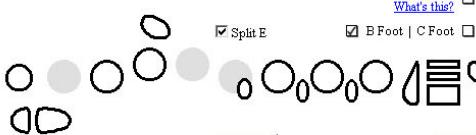
© Andrew Botros 2001-2005
abotros@phys.unsw.edu.au

★ [What's new!](#) [Your comments](#) [How To Guide](#) [Credits](#)

1. Click the keys of a fingering to search for all its predicted notes and multiphonics ... [?](#)
[A note on flute models](#)

Allow unconventional finger positions [What's this?](#)

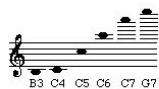
Split E B Foot | C Foot



Search **Clear**

2. Select a note from the list to search for an alternate fingering, trill or microtone ... [?](#)
e.g. A4, C#6. Use [standard note names](#). [F#7](#) and [G7](#)

SELECT NOTE



C	-	3
D	#	4
E		5
F		6
G		7
A		
B		

B Foot C Foot Both

Split E No split E Both

Allow unconventional finger positions [What's this?](#)

[More on notes and keys...](#)

Fingerings MUST include the keys... And must NOT include the keys...

<input type="checkbox"/> BbTh or Th	<input type="checkbox"/> G	<input type="checkbox"/> D#
<input type="checkbox"/> 1	<input checked="" type="checkbox"/> F#	<input type="checkbox"/> C#
<input type="checkbox"/> A#	<input type="checkbox"/> 1	<input type="checkbox"/> C
<input type="checkbox"/> 2	<input type="checkbox"/> 2	<input type="checkbox"/> B
<input type="checkbox"/> 3	<input type="checkbox"/> 3	<input type="checkbox"/> gizmo
<input type="checkbox"/> G#	<input type="checkbox"/> tr1	
	<input type="checkbox"/> tr2	
	<input type="checkbox"/> Bblever	

<input type="checkbox"/> BbTh or Th	<input type="checkbox"/> G	<input type="checkbox"/> D#
<input type="checkbox"/> 1	<input type="checkbox"/> F#	<input type="checkbox"/> C#
<input type="checkbox"/> A#	<input type="checkbox"/> 1	<input type="checkbox"/> C
<input type="checkbox"/> 2	<input type="checkbox"/> 2	<input type="checkbox"/> B
<input type="checkbox"/> 3	<input type="checkbox"/> 3	<input type="checkbox"/> gizmo
<input type="checkbox"/> G#	<input type="checkbox"/> tr1	
	<input type="checkbox"/> tr2	
	<input type="checkbox"/> Bblever	

OR... Include any keys OR... Don't exclude any keys

Search

3. Search for a multiphonic fingering ... [?](#)
e.g. C5&D6. Use [standard note names](#).

Select two or possibly three different notes of a multiphonic,
or select only one note to search for multiphonics which include that note:

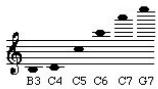
SELECT NOTE 1 **SELECT NOTE 2** **SELECT NOTE 3**

C	-	3
D	#	4
E		5
F		6
G		7
A		
B		

C	-	3
D	#	4
E		5
F		6
G		7
A		
B		

C	-	3
D	#	4
E		5
F		6
G		7
A		
B		

Clear Notes



B Foot C Foot Both

Split E No split E Both

Allow unconventional finger positions [What's this?](#)

[More on notes and keys...](#)

Search

SIEMENS WINNER SIEMENS PRIZE FOR INNOVATION 2002  WINNER AUSTRALIAN ACOUSTICAL SOCIETY EXCELLENCE IN ACOUSTICS AWARD 2003

Figure 1.2: The front page of The Virtual Flute (Botros et al. 2006).

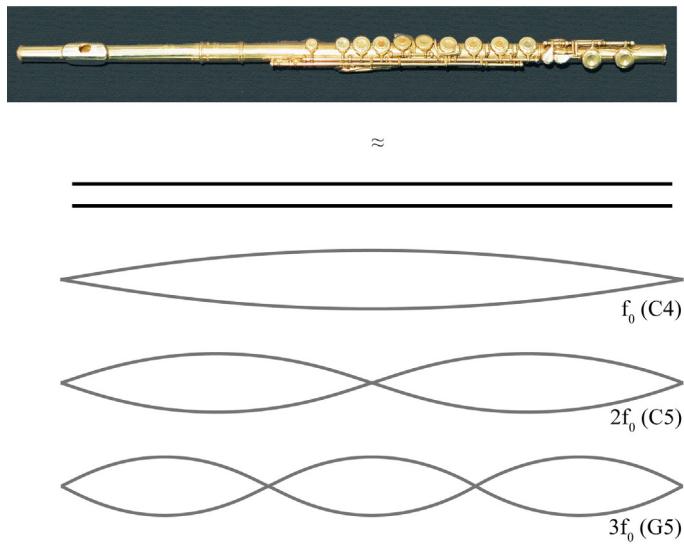


Figure 1.3: The flute may be approximated as a pipe open at both ends. Pressure modes corresponding to the first three harmonics are shown.

several dozen fingerings without hogging the flute maker's computer and wasting too much of his time, and accessed through a user-friendly interface, with most of the physics removed from view.

1.4 HOW FLUTES WORK

A brief introduction to the relationship between music and physics is given in Appendix C. To a physicist, the zeroth order model of a flute is that of a pipe open at both ends. When a jet of air is directed across one of the open ends at the appropriate speed, pulses of air set up a standing wave within the pipe. Some of the energy from the standing wave escapes and propagates to the ear, where it is perceived as sound. Holes in the side of the flute can be opened or closed by the player's fingers, which changes the effective length of the pipe and the frequency of the notes that can be played.

A flute with all the tone holes closed has a long column of air, open at both ends. When the flute is played, sound waves travel up and down the flute, producing a standing wave. The air column in the flute resonates at particular frequencies—these different *modes* determine the notes that are possible. The first three modes for all holes closed are shown in Figure 1.3. These produce the notes C4 (middle C), C5 and G5.

A flutist plays a flute by directing a stream of air across the embouchure hole. This creates an acoustic disturbance that propagates the length of the flute, is reflected back by the open end and interacts again with the jet. If the length and speed of the jet are such that oscillations of the jet are in phase with a resonance of the air column, a sustained note is produced.

1.5 ACOUSTIC IMPEDANCE: PREDICTING THE PLAYING QUALITIES OF A FLUTE

Many of the acoustical characteristics of a flute may be quantified by its *acoustic impedance* spectrum. Acoustic impedance is the ratio of pressure to flow at the input to the instrument and describes how the flute will ‘respond’ to excitation at a particular frequency. The magnitude of the acoustic impedance of a flute varies over the frequency range of the instrument and has many maxima and minima. Because the flute is open to the air at the embouchure, the minima correspond to the possible standing waves, and the notes the flute can play are close in frequency to these minima. The impedance spectrum can also tell us about the tonal characteristics and playability of a note. Generally, if for some fingering there are many impedance minima with frequencies in a harmonic relationship to each other, then the note produced will be harmonically rich (bright); if not the note will be more pure (or dull sounding). Playability of a note depends upon many factors, such as how deep its impedance minima are, and whether or not there are other minima nearby. Each fingering for the flute produces a different impedance curve, with different possible standing waves.

Many methods may be used to measure the acoustic impedance of woodwind instruments. At least, some source of acoustic energy is needed (such as a loudspeaker) along with a means of measuring or inferring both the pressure and the flow. A review of many methods that have been used is provided in Chapter 3, which also documents the development of a novel, high-precision measurement and calibration technique.

The acoustic impedance of a flute can be modelled by treating the flute as a one-dimensional network of cylinders and cones. We may then calculate the impedance properties of each element and derive the impedance of the entire flute. Figure 1.4 illustrates such an approach. Starting at the downstream end of the flute, the impedance of the last cylindrical segment is calculated (Z_1), as is the impedance of the last tone hole (Z_2). These impedances are combined in parallel and the result used in the calculation of the impedance at the next section. The impedance is calculated in an iterative manner until the entire flute is modelled. If the geometrical lengths of the flute segments are used without correction to calculate the impedance, it will not be accurate. This is because of various effects that are not adequately modelled by the one-dimensional network described above. However, adequate results can be achieved if measured length corrections are used. The tone hole segments (but not the bore) in Figure 1.4 are shown with length corrections added.

1.6 TUNING, TRADITION AND THE ‘STANDARD FLUTIST’

Musicians can be (understandably) resistant to people meddling with their instruments, and a flute judged to be perfectly in tune by one person may be out of tune to another. Musical instruments usually evolve, as instrument makers adapt to changing musical tastes and playing styles. Musicians are taught to compensate for their instrument’s peculiarities, and with years of practice they do this remarkably well. For this reason it is often difficult to convince musicians that a new design is better than the existing one, since adopting the ‘improved’ instrument will entail ‘un-learning’ some of their technique. Furthermore, the concept of an

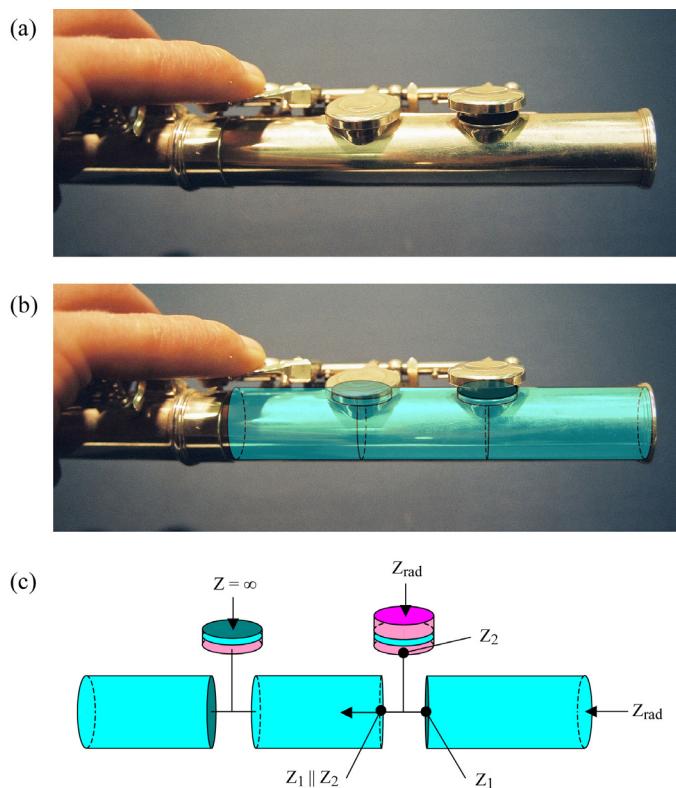


Figure 1.4: The footjoint of a modern flute showing one closed hole and one open hole (a). The impedance of the flute is modelled by dividing the flute into cylindrical and conical segments (b) and adding length corrections (c).

‘in-tune’ instrument is difficult to define.

A flute plays notes close in frequency to the impedance minima in its impedance spectrum. How close depends on several factors, including the impedance of the player’s face and embouchure, and the frequency of other impedance minima. In order to approximate the frequency-dependence of a typical player’s embouchure, we asked professional and semi-professional flute players to play notes on the flute without aiming to play the note in tune, but rather to aim for the best sound. We then compared the tunings obtained in this way to those measured on the impedance spectrum, and used the difference in a correction factor.

1.7 SOFTWARE IMPLEMENTATION: GOALS AND LIMITATIONS

What are the requirements of the software program that is the practical aim of this thesis?

The maker should be able easily to enter, store and retrieve flute data in a familiar form. The interface should additionally show a schematic diagram of the instrument for a quick overview of the flute design and for comparison to existing designs.

The software should calculate and display the acoustic parameters of interest (tuning, timbre and playability) for a couple of dozen important notes reasonably quickly—within a minute at most on an inexpensive computer. This condition is important, since the flute maker should be able to make iterative changes to a design and see the effect of each change without having to wait an inordinate amount of time. This limits the software models available to us and has implications for the software implementation. It would be useful as a design aid if the software has the facility to plot the magnitude of the pressure and flow waves along the flute for any fingering, as these give insight into the contribution of each tone hole to the tuning of a particular note and help with the correct placement of register holes.

The software program C was used to produce the impedance model. Some of the code used in the impedance model of Botros et al. (2006) is inherited and used in this thesis. The command line input to the C program is an XML file describing the flute geometry, and a file containing fingering information. The program outputs the impedance spectra for each fingering. A related software tool outputs the pressure and flow magnitudes along the flute.

The software interface was written using Java. Java is an object-oriented programming language developed by Sun Microsystems. One of the main advantages of Java is that it is platform independent: provided the user has a Java Runtime Environment (JRE) installed on their machine, the code will run. JREs are available for all major operating systems.

1.8 GUIDE TO THE THESIS

In Chapter 2 the relevant literature is surveyed, important formulae are quoted or derived and existing flute models are compared. Next, an impedance spectrometer is made and carefully calibrated (Chapter 3) so as to measure the input impedance of wind instruments quickly and with a high degree of accuracy. The impedance characteristics of finger holes, such as are used on recorders, classical flutes and clarinets, are measured in Chapter 4 and fit-formulae are derived. Databases of impedance spectra for modern and classical flutes and a modern clarinet are assembled (Chapter 5). Possible effects on impedance spectra of the material used for flute

construction are discussed in Chapter 6 and the effects of the player's embouchure and face are measured in Chapter 7. The findings in these early chapters form the basis of the flute impedance model, which is implemented in the computer language C and a user-friendly graphical interface to the model is created using the computer language Java (Chapter 8). Finally, several applications of the software package are discussed (Chapter 9) and further directions for the work are explored.

Chapter II

Theory and literature review

This chapter gives a brief overview of the theory and an introduction to the literature relevant to this project. Firstly, the history of flutes and flute making is covered, and some ideas are offered as to where flute design may be headed. The theory of woodwind acoustics is then reviewed, and equations are presented where appropriate. Finally, several woodwind models reported in the literature are discussed and compared with regards to the expected outcomes of this project.

2.1 FLUTES AND FLUTE MAKING

Modern instruments in the extensive flute family include the modern orchestral transverse flute, the baroque transverse flute, the recorder, flue organ pipes and the Japanese shakuhachi (Baines 1967). Sound is produced in these instruments when a stream of air interacts with a resonating body. This section briefly reviews the history of flutes and flute making, with particular reference to the transverse flute from the baroque period onwards.

2.1.1 History

Flutes are the oldest known musical instruments and have been made at least since Palaeolithic times (Dauvois et al. 1998). The earliest flutes were made of bone. Most of these early flutes were close to cylindrical, with up to six finger holes. Figure 2.1 shows a picture of an early bone flute, along with a baroque flute, a classical flute and a modern flute.

The baroque flute was developed in the second half of the 17th century (Bate 1975), and included two important innovations—the bore was made conical rather than cylindrical (with the largest diameter in the section near the embouchure hole) and a single key was added. The conical bore improved the intonation, and the key made the flute fully chromatic. As a result of these changes, the flute gained in popularity, particularly as a solo instrument. On a baroque instrument the notes in the diatonic scale are produced with simple fingerings (consisting of an array of closed tone holes followed by an array of open tone holes) and most other notes are made with cross fingerings or the use of the key. Consequently, considerable variation exists in the timbre, volume and ease of playing across the chromatic scale.

The flute continued to develop through the classical period, as more keys were added and the holes were enlarged. Makers at this time generally sought more volume, a ‘stronger tone’ and greater uniformity of timbre among different notes. On a classical flute, fewer cross fingerings are used, and are necessary only in the third octave of the instrument.

Most flutes of the baroque and classical periods play a natural major scale based on the note D4. Some classical flutes have keys for the notes C4 and C \sharp 4, and some are designed to play in different keys (such as B \flat or E). The piccolo is a small flute that generally plays an octave



Figure 2.1: An early bone flute (a), a baroque flute (b), a classical flute (c) and a modern flute (d).

higher than an ordinary flute.

The modern flute was developed in the mid-19th century with the largest developments made by Theobald Boehm (Boehm 1871). Boehm redesigned the flute bore, making it mostly cylindrical with a diameter of 19 mm, and tapering the bore to around 17 mm at the cork. Boehm also aimed to place the holes at their ‘acoustically correct’ positions, and developed a system of keys and clutches that allows the player to play most of the notes in the equal tempered chromatic scale with no cross fingerings. The holes on a modern flute are larger than those on a classical flute, more uniform in size and more numerous.

Most modern flutes are made of metal although many early flutes were made to Boehm’s design with a wooden bore and silver keys (Fletcher & Rossing 1998). Flutes from the 19th century exist that use Boehm’s key system with a conical bore. Modern orchestral flutes differ very little from Boehm’s design.

The history of the flute is discussed by Rockstro (1890), Baines (1967) and Bate (1975). Wolfe et al. (2001a) give an overview of the acoustics of flutes and Wolfe & Smith (2003) compare the use of cross fingerings in baroque, classical and modern flutes. Fabre & Hirschberg (2000) review physical models of the flute excitation mechanism. Terry McGee’s web site <<http://www.mcgee-flutes.com>> provides extensive information about flute making and the history of classical flutes.

2.1.2 Materials for flute making

Many different materials have been used to make flutes, including wood, ivory, glass, ebonite and metal (Rockstro 1890). Of the various types of wood available, flute makers favour fine-grained, dense timbers, such as cocuswood, African blackwood or European boxwood. Many timber species traditionally used to make flutes are rare or endangered, and modern makers are increasingly looking to different species for sustainable production. Terry McGee, for example,

offers flutes in Australian timber species such as gidgee and Cooktown ironwood. Timber to make flutes needs to be geometrically stable and durable, relatively impervious to moisture, and capable of being worked to a smooth finish.

High quality modern flutes are made from precious metals such as silver, gold and platinum, a common conception being that more costly materials produce better flutes. The least expensive flutes are made from nickel-silver (a copper-nickel-zinc alloy) which is then silver plated.

Argument has raged over the relative merits of flutes made from various materials. Coltman (1971) presented a series of experiments where he made three identical keyless flutes from silver, copper and grenadilla wood. The three flute bodies were attached to identical headpieces made from Delrin plastic, and the geometrical properties of the flutes were identical to within 0.1 mm. When the three flutes were played to two groups of listeners, one chosen as being ‘musically skilled’, the other ‘unskilled’, neither group could distinguish between the instruments. In a related experiment flutists were asked to play the flutes and were unable to distinguish between the different materials. In a more recent study Widholm et al. (2001) used listening tests to compare flutes made from many different metals and metal alloys. The findings of Widholm et al. were similar to those of Coltman.

Clearly, materials which are highly porous or cannot be machined sufficiently smooth will produce instruments of inferior quality. In Chapter 6, some experiments are presented showing the effect of humidity and oiling on the impedance of timber bores. As part of this study, Terry McGee made a flute from radiata pine (a light, highly porous timber, much more suited to making a case for the flute than the instrument itself). The results are discussed in Chapter 6.

2.1.3 Contemporary flute design

The modern flute has changed little since its invention. Refinements continue to be made but these are hardly equivalent to the changes to flute making that Boehm initiated in the 19th century. Various makers have championed different schemas for the size and placement of holes, the eponymous ‘Cooper scale’ being a prominent example. Meanwhile, makers such as Terry McGee maintain that the modern flute is not well suited to many types of music, including Irish folk music and classical music, and believe that there should be several divergent lines of flute development. Musicians involved in the ‘original instrument movement’ use period instruments (or good copies) to play music by composers such as Bach and Mozart, in an effort to recreate the exact sound heard or imagined by those composers.

Niche markets exist for designing flutes for particular styles of music. For example, players in the Irish and Cuban traditions use, respectively, classical and romantic flutes, but neither is optimised for the requirements of that particular style. Contemporary composers increasingly use multiphonics and microtones. The Virtual Flute (Botros et al. 2006) searches all possible fingerings on the modern flute for suitable fingerings, but in the future flutes may be designed with such requirements explicitly considered so that composers need no longer be limited to what is available through chance.

2.2 ACOUSTICS OF WOODWIND INSTRUMENTS

The acoustics of woodwind instruments is treated extensively in several reference works (Bennade 1960, Nederveen 1998, Fletcher & Rossing 1998). This section is not intended to be nearly as exhaustive as those works, and the interested reader is referred to them for more information. This section will give a brief coverage of the acoustics of woodwind instruments, with particular reference to the calculation of the input impedance.

2.2.1 Properties of air

The speed of sound in air c is approx. 343 m s^{-1} at 20°C and 50% relative humidity. The speed of sound increases with the square root of absolute temperature, and with water vapour content. Carbon dioxide in air decreases the speed of sound. Cramer (1993) gives a formula for the speed of sound under various conditions of temperature, pressure, humidity and carbon dioxide content.

The density of air ρ is approx. 1.20 kg m^{-3} at 20°C (dry air) and decreases with both temperature and water vapour content. Giacomo (1982) gives a formula for the density of moist air with variable carbon dioxide content.

Coltman (1966) discusses the changes in the speed of sound caused by a flute player's breath. The effect of the player's breath on the speed of sound is threefold. The speed of sound changes as a result of changes in the temperature, the carbon dioxide content of the air, and the amount of water vapour in the air. The latter two effects tend to cancel each other to an extent; carbon dioxide being denser than air and water vapour less dense. Coltman (1966) finds that the carbon dioxide content in the air is approximately 2.5%, enough to flatten the flute resonances by 12 cents.

Coltman (1968a) measured the temperature at different points along the modern flute under playing conditions. The data can be reasonably well fitted by the equation $T = 30.3 - 7.7x$, where T is the temperature in degrees Celsius and x is the distance along the flute in metres from the embouchure. While these data are for the modern flute, the derived equation can probably be applied to any geometry, and in both directions from the embouchure hole.

2.2.2 Acoustic pressure and flow

The frequency range of human perception of sound is from 20 Hz to 20 kHz. Air pressure variation within this frequency range is termed acoustic pressure $p = P - P_0$, where P and P_0 are respectively the instantaneous and atmospheric pressures. Acoustic pressure is measured in pascals (Pa). Due to the large dynamic range of the human ear, acoustic pressure is often expressed in decibels (dB) where

$$\text{dB} = 20 \log_{10} \left(\frac{p}{p_0} \right) \quad (2.1)$$

and $p_0 = 20 \mu\text{Pa}$, chosen because it is close to the *threshold of hearing* at optimal frequencies.

The time-varying airflow associated with acoustic pressure changes is referred to as acoustic flow u with units of velocity (m s^{-1}). It is often more convenient to use acoustic volume flow U which has units of $\text{m}^3 \text{s}^{-1}$.

At any point x along a one-dimensional duct, the acoustic energy in a volume element with

infinitesimal width dx and area S is $E(x) dx$ where the energy function $E(x)$ is given by

$$E(x) = \frac{1}{2} \left(\frac{p^2 S}{\rho c^2} + \frac{\rho U^2}{S} \right). \quad (2.2)$$

2.2.3 Wave propagation in pipes and horns

The acoustic pressure wave inside wind instruments with rigid walls can be determined generally by solving the wave equation

$$\nabla^2 p = \frac{1}{c^2} \frac{\partial^2 p}{\partial t^2} \quad (2.3)$$

subject to the boundary condition that $\mathbf{n} \cdot \nabla p = 0$ at the instrument walls, where \mathbf{n} is a unit vector normal to the wall. Simple solutions to this equation exist for several geometries, including cylindrical tubes and conical horns (Fletcher & Rossing 1998). Approximate solutions can be found for other geometries, such as the exponential horn. The acoustic flow velocity u is related to the pressure according to Newton's law

$$u = \frac{i}{\rho \omega} \nabla p \quad (2.4)$$

where $\omega = 2\pi f$ is the angular frequency.

The solution to (2.3) in cylindrical polar coordinates for a cylindrical pipe of radius a is proportional to $\exp[i(-k_{mn}x + \omega t)]$, where the proportionality depends on r and ϕ and describes how the pressure wave is distributed over the (r, ϕ) plane. The square of the wave vector k_{mn} for the mode (m, n) is given by

$$k_{mn}^2 = \left(\frac{\omega}{c} \right)^2 - \left(\frac{\pi q_{mn}}{a} \right)^2, \quad (2.5)$$

where m and n are mode numbers corresponding to the numbers of nodal diameters and nodal circles in the (r, ϕ) plane and q_{mn} is defined by the boundary condition that there can be no flow in the radial direction at the cylinder walls.

The (m, n) mode propagates (extends infinitely in the x direction, neglecting any losses) if k_{mn} is real; otherwise the acoustic pressure decays exponentially with x and the mode is termed *evanescent*. The plane wave mode given by $m = n = 0$ will always propagate since $q_{00} = 0$; higher modes will only propagate at frequencies above the cutoff frequency for the mode given by

$$\omega_c = \frac{\pi q_{mn} c}{a}. \quad (2.6)$$

The higher mode with the lowest cutoff frequency is the $(1, 0)$ mode, which has a single nodal plane. For this mode

$$\omega_c = \frac{1.83c}{a}, \quad (2.7)$$

giving a cutoff frequency of approx. 10 kHz for a typical flute bore radius of 10 mm.

Similar equations may be derived for conical horns although the cutoff frequency for each mode varies with radius along the length of the horn, and the wavefronts for the always-propagating mode are spherical rather than planar.

Fortunately, in the bores of most wind instruments all higher modes are non-propagating at frequencies of musical interest. Higher modes are evoked in the vicinity of bore disturbances, such as discontinuities or tone holes, but decay quickly. The fields around tone holes may be

decomposed into the various modes, but often the plane wave or spherical wave approximations provide a sufficiently accurate description of the behaviour of an instrument, provided length or impedance corrections are applied at points where the plane wave approximation breaks down.

2.2.4 Acoustic impedance

The acoustic impedance Z at any point in an acoustic field is defined as the ratio of pressure to volume flow. For a plane wave propagating in an infinite cylindrical pipe, the acoustic impedance is independent of frequency, having the value

$$Z_0 = \frac{\rho c}{S}, \quad (2.8)$$

where S is the cross-sectional area of the pipe. Z_0 is known as the characteristic impedance of the pipe. For finite length pipes and cones, the plane-wave acoustic impedance is a function of frequency; formulae for various situations are given below. The acoustic impedance is related to the reflection coefficient R (the amplitude of the reflected wave relative to that of the incident wave) according to the relation

$$Z = Z_0 \frac{1+R}{1-R}. \quad (2.9)$$

A review of techniques for impedance measurement is given in Chapter 3 and the acoustic input impedance of flutes and clarinets is discussed at length in Chapter 5.

2.2.5 Plane waves in cylindrical pipes

The plane wave mode in a lossless cylindrical pipe will always propagate with wavenumber $k = \frac{\omega}{c}$ and pressure given by

$$p(x, t) = [Ae^{-ikx} + Be^{ikx}]e^{i\omega t}, \quad (2.10)$$

where A and B are the amplitudes of the forward- and backward-going waves respectively. The acoustic volume flow is given by

$$U(x, t) = \left(\frac{S}{\rho c} \right) [Ae^{-ikx} - Be^{ikx}]e^{i\omega t}. \quad (2.11)$$

For real pipes with non-negligible wall losses k is complex, and the amplitude of the wave decreases exponentially as it propagates (see §2.2.7).

The transfer matrix \mathbf{T} for a cylindrical pipe of length L relates the pressure and volume flow at the input to those at the output (Figure 2.2). Using (2.10) and (2.11), it can be shown that

$$\begin{bmatrix} p_1 \\ U_1 \end{bmatrix} = \mathbf{T} \begin{bmatrix} p_2 \\ U_2 \end{bmatrix} \quad (2.12a)$$

where

$$\mathbf{T} = \begin{bmatrix} \cosh(i k L) & Z_0 \sinh(i k L) \\ \frac{1}{Z_0} \sinh(i k L) & \cosh(i k L) \end{bmatrix}. \quad (2.12b)$$

The input impedance of a cylindrical pipe section terminated by a load impedance Z_L may be derived from (2.12) by multiplication after setting $p_2 = Z_L U_2$. This gives

$$Z_{IN} = Z_0 \left[\frac{Z_L \cosh(i k L) + Z_0 \sinh(i k L)}{Z_L \sinh(i k L) + Z_0 \cosh(i k L)} \right]. \quad (2.13)$$

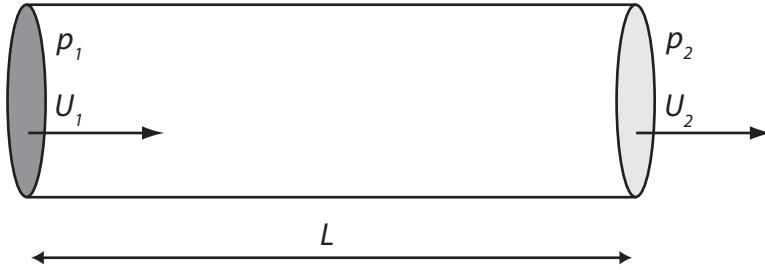


Figure 2.2: Pressure and volume flow for a cylindrical pipe element of length L . The transfer matrix relates the acoustic quantities at the input (subscript 1) to those at the output (subscript 2).

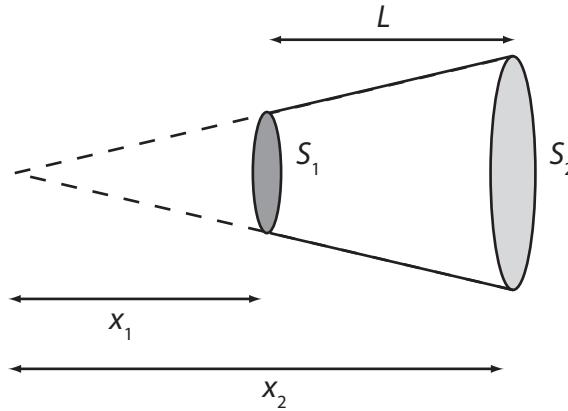


Figure 2.3: A truncated cone with throat area S_1 and mouth area S_2 .

Two special cases of (2.13) are worth considering. If the output of the pipe is rigidly closed, $Z_L = \infty$ to a very good approximation and

$$Z_{\text{IN}}^{\text{stopped}} = -iZ_0 \cot(kL). \quad (2.14)$$

On the other hand, if the output is ideally open with $Z_L = 0$,

$$Z_{\text{IN}}^{\text{open}} = iZ_0 \tan(kL). \quad (2.15)$$

No real pipe is ideally open, as the air outside the pipe presents a non-zero impedance. Equations for calculating the radiation impedance of the open end are given in §2.2.8.

2.2.6 Spherical waves in conical horns

The pressure wave in a conical horn is proportional to $\exp[i(-kx + \omega t)]$, with an amplitude inversely proportional to the radius of the cone. Here x is measured from the apex of the cone and the wavefronts are understood to be spherical. However for low cone angles, the plane-wave approximation may be used.

For the truncated cone with dimensions shown in Figure 2.3, i.e. a throat of area S_1 , a mouth of area S_2 , length L , and distances of the throat and mouth from the conical apex x_1 and x_2 , the transfer matrix is given by

$$\mathbf{T} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad (2.16a)$$

with

$$A = -\frac{\sin(kL - \theta_2)}{\sin\theta_2} \quad (2.16b)$$

$$B = i \frac{\rho c}{S_2} \sin(kL) \quad (2.16c)$$

$$C = i \frac{S_1}{\rho c} \frac{\sin(kL + \theta_1 - \theta_2)}{\sin\theta_1 \sin\theta_2} \quad (2.16d)$$

$$D = \frac{S_1}{S_2} \frac{\sin(kL + \theta_1)}{\sin\theta_1}, \quad (2.16e)$$

where $\theta_1 = \tan^{-1}(kx_1)$ and $\theta_2 = \tan^{-1}(kx_2)$. Fletcher & Rossing (1998) give expressions for the input impedance of conical horns with various load impedances. Fletcher & Rossing also treat other horn geometries, such as the exponential horn—these results are not reproduced here.

2.2.7 Wall losses

So far in our discussion we have assumed that the duct walls are perfectly smooth, rigid and thermally insulating, and that acoustic waves propagate without loss along the duct. For most woodwinds the walls are sufficiently rigid that mechanical vibration of the walls can be safely neglected*. However, waves within real woodwind instruments are subject to thermal dissipation and viscous drag, leading to attenuation of the travelling wave.

As discussed by Nederveen (1998), the acoustic flow slows to zero near the tube walls, and thermal exchange between the walls and the air removes some energy from the (adiabatic) acoustic waves. For woodwind instruments, both of these effects occur only over a thin boundary layer (approx. 0.05 mm at 1000 Hz in the case of the viscous effects), so the plane-wave approximation remains valid (Nederveen 1998).

The effects of viscous drag and thermal exchange are discussed by Benade (1968). In general, the effect of loss terms is to make Z_0 and k complex, which leads to attenuation of the wave as it travels along the conduit. The small imaginary part of Z_0 can usually be neglected for the bores of musical instruments. The complex wavenumber is given by

$$k = \frac{\omega}{v} - i\alpha, \quad (2.17a)$$

where

$$\nu = c \left[1 - \frac{1}{r_v \sqrt{2}} - \frac{(\gamma - 1)}{r_t \sqrt{2}} \right] = c \left[1 - \frac{1.65 \times 10^{-3}}{af^{1/2}} \right] \quad (2.17b)$$

and

$$\alpha = \frac{\omega}{c} \left[\frac{1}{r_v \sqrt{2}} + \frac{(\gamma - 1)}{r_t \sqrt{2}} \right] = \frac{3 \times 10^{-5} f^{1/2}}{a}. \quad (2.17c)$$

Here r_v and r_t are the ratios of pipe radius to boundary layer thickness for viscous and thermal losses respectively. Approximations for these ratios are given in Fletcher & Rossing (1998). Equations (2.17b) and (2.17c) are valid for $r_v > 10$. As shown by Keefe (1984) the viscothermal effects on the characteristic impedance Z_0 become significant for $r_v < 10$, as do terms of higher order in r_v and r_t in the expressions for the phase velocity (2.17b) and attenuation coefficient

*According to measurements by Backus (1964) and Nederveen (1998), the walls of woodwind instruments vibrate with an amplitude of about 1 μm and radiation due to this vibration is at least 40 dB smaller than the air signal.

(2.17c). The condition $r_v = 10$ corresponds to a tube diameter of approx. 1 mm. Since this is much smaller than typical bores or tone holes used in flutes, the simplified expressions given above have been used in this thesis.

2.2.8 Radiation impedance

The impedance at the end of an open pipe is small but not negligible, and depends on the frequency, the extent and geometry of flanging around the open end and the presence of any occluding objects (such as an overhanging keypad or a player's finger). In this section I will compare various theoretical and experimental determinations of the radiation impedance for the following cases:

- an unflanged pipe
- an infinite flange
- a circular flange
- a cylindrical flange
- a disk poised over a circular flange.

The real and imaginary parts of the radiation impedance for both flanged and unflanged pipes are shown in Figure 2.4, in terms of the dimensionless quantity ka , where a is the radius of the pipe. In the low frequency limit $ka \ll 1$, the impedance is purely reactive and equal to the impedance of a short, ideally open section of pipe. This leads to the concept of a length correction. To calculate the impedance of an open pipe, it is sufficient at low frequencies to calculate the impedance of a slightly longer pipe with a zero load impedance. For higher frequencies the length correction must be allowed to vary with frequency to give a true representation of the radiation impedance. The small imaginary part of the end correction (due to radiation losses from the end of the pipe) must also be included. Following Dalmont et al. (2001), we shall denote the frequency-dependent end correction by $\tilde{\delta}$ and the complex, frequency dependent end-correction by $\tilde{\delta}^*$.

The complex, frequency dependent end correction is related to the real, frequency dependent end correction and to the modulus of the reflection coefficient according to

$$\tilde{\delta}^* = \tilde{\delta} + i \frac{\ln(|R|)}{2k} \quad (2.18)$$

and the impedance is just the impedance of an ideally open pipe of length $\tilde{\delta}^*$:

$$Z = iZ_0 \tan(k\tilde{\delta}^*). \quad (2.19)$$

2.2.8.1 Unflanged pipe

The low-frequency, lossless end correction for an open unflanged pipe is (Dalmont et al. 2001)

$$\delta_{\text{open}} = 0.6133a. \quad (2.20)$$

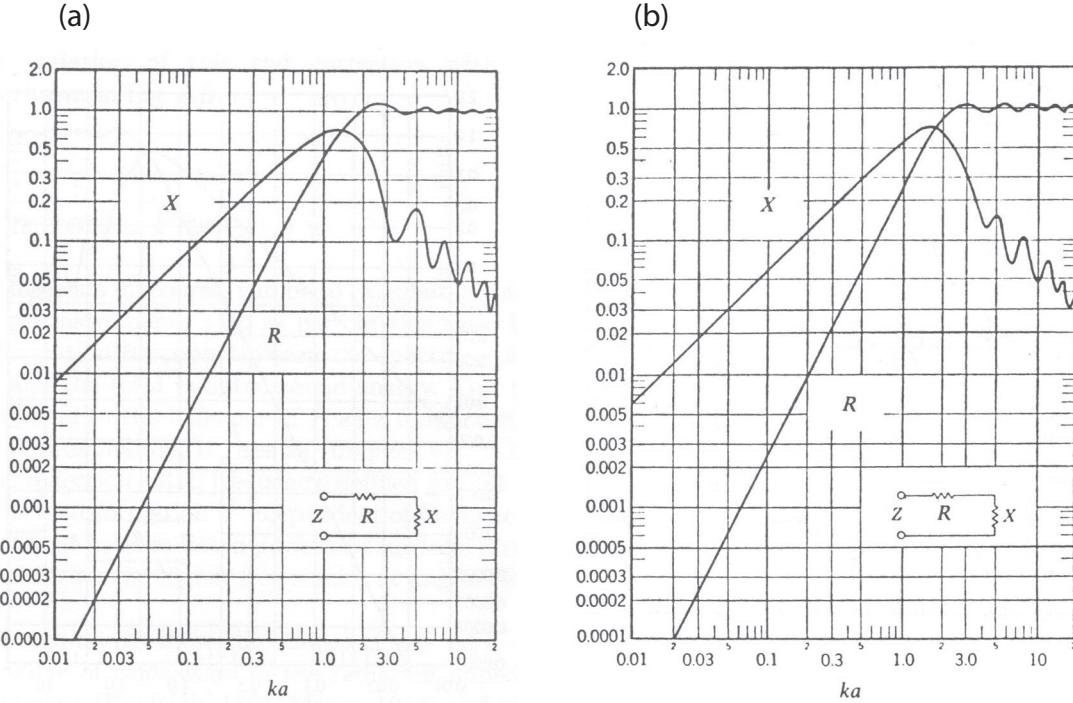


Figure 2.4: The acoustic reactance X and the acoustic resistance R as functions of ka for an infinitely flanged pipe (a) and an unflanged pipe (b). X and R are both in units of $\rho c / \pi a^2$ (Fletcher & Rossing 1998, after Beranek (1954)).

The impedance (including radiation loss) of an unflanged pipe was calculated by Levine & Schwinger (1948). Dalmont et al. (2001) give simple fit formulae for the frequency-dependent end correction and the magnitude of the reflection coefficient:

$$\tilde{\delta}_{\text{open}} = \delta_{\text{open}} \left[\frac{1 + 0.044(ka)^2}{1 + 0.19(ka)^2} - 0.02 \sin^2(2ka) \right] \text{ for } ka < 1.5 \quad (2.21)$$

and

$$|R_{\text{open}}| = \frac{1 + 0.2ka - 0.084(ka)^2}{1 + 0.2ka + (\frac{1}{2} - 0.084)(ka)^2} \text{ for } ka < 3.5. \quad (2.22)$$

2.2.8.2 Infinitely flanged pipe

The low-frequency, lossless end correction for an infinitely flanged pipe is greater than for an unflanged pipe, since the solid angle for radiation is reduced (Dalmont et al. 2001):

$$\delta_{\text{flanged}} = 0.8216a. \quad (2.23)$$

Rayleigh (1894) and Nomura et al. (1960) give the impedance for an infinitely flanged pipe. Fit formulae by Norris & Sheng (1989) are accurate for $ka < 3.5$:

$$\tilde{\delta}_{\text{flanged}} = \delta_{\text{flanged}} \left[1 + \frac{(0.77ka)^2}{1 + 0.77ka} \right]^{-1} \quad (2.24)$$

and

$$|R_{\text{flanged}}| = \frac{1 + 0.323ka - 0.077(ka)^2}{1 + 0.323ka + (1 - 0.077)(ka)^2}. \quad (2.25)$$

2.2.8.3 Circular flange

The end correction for a circular flange of finite radius has been investigated experimentally (Benade & Murday 1967) and analytically (Ando 1970, Bernard & Denardo 1996). Dalmont et al. (2001) compared the results in the literature and obtained the following fit formula from numerical calculations using the finite difference method:

$$\delta_{\text{circ}} = \delta_{\text{flanged}} + \frac{a}{b}(\delta_{\text{open}} - \delta_{\text{flanged}}) + 0.057 \frac{a}{b} \left[1 - \left(\frac{a}{b} \right)^5 \right] a, \quad (2.26)$$

where b is the radius of the flange. This result was obtained for the low frequency limit, but is assumed to apply also to complex, frequency-dependent end corrections. In this case, an extra term must be added to the reflection coefficient to account for reflections that arise at the edge of the flange. Then

$$R_{\text{circ}} = R_{\text{noref}} + R_{\text{edge}} \quad (2.27)$$

where R_{noref} is calculated from the end correction given by (2.26) and

$$R_{\text{edge}} = -0.43 \frac{(b-a)a}{b^2} \sin^2 \left(\frac{kb}{1.85 - a/b} \right) e^{-ikb[1+a/b(2.3-a/b-0.3(ka)^2)]}. \quad (2.28)$$

This equation is valid for $a/b \geq 0.2$, $ka < 1.5$ and $kb < 3.5$. The radiation impedance is related to the reflection coefficient by (2.9).

2.2.8.4 Cylindrical flange

Open tone holes drilled through the wall of a woodwind instrument bore are flanged by the (approximately) cylindrical outside wall of the instrument. Dalmont et al. (2001) used finite difference methods to calculate the low frequency end correction for a pipe (of radius a) flanged by a cylinder (radius b). Dalmont et al. proposed the following fit-formula for $a/b < 0.7$:

$$\delta_{\text{cyl}}/a = \delta_{\text{flanged}}/a - 0.47(a/b)^{0.8}. \quad (2.29)$$

Dalmont et al. also estimated δ_{cyl} by measuring the impedance of flanged tubes. The experimentally-determined value for δ_{cyl} was significantly greater than that derived from the finite difference method for $a/b < 0.5$, the difference being approx. 10% for $a/b = 0.25$. Earlier, Benade & Murday (1967) found $\delta_{\text{cyl}} = 0.64a[1+0.32\ln(0.3b/a)]$ for $0.14 \leq a/b \leq 0.67$, by measuring the resonances of pipes with holes drilled through the wall. The formulae of Dalmont et al. and Benade & Murday are compared in Figure 2.5.

2.2.8.5 Disk poised over circular flange

A keypad hanging over a tone hole increases the radiation impedance significantly. Simulations by Dalmont et al. (2001) using the finite difference method found that a disk poised above a tube with circular flange increased the end correction by

$$\delta_{\text{disk}} - \delta_{\text{circ}} = \frac{a}{3.5(h/a)^{0.8}(h/a + 3w/a)^{-0.4} + 30(h/d)^{2.6}}, \quad (2.30)$$

where a is the radius and w the wall thickness of the tube, d is the radius of the disk and h the height of the disk above the end of the tube. The thickness of the disk itself was found to

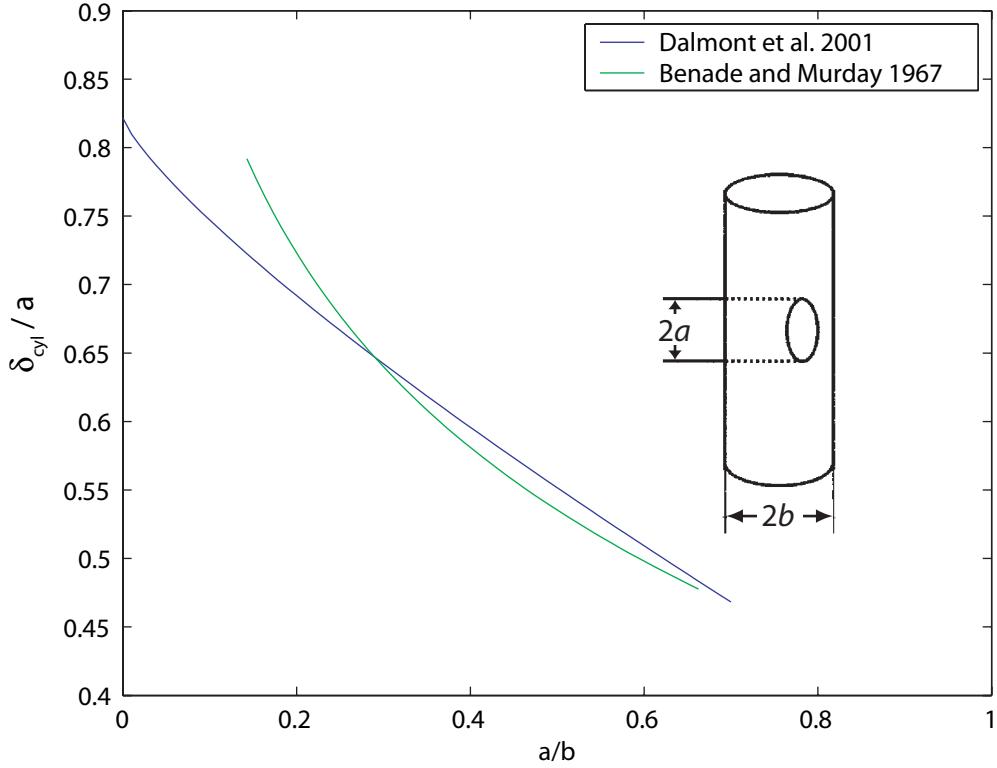


Figure 2.5: The dimensionless end correction δ_{cyl}/a for a cylindrical pipe flanged by a cylinder as a function of radius ratio.

have negligible effect. Equation (2.30) is applicable over the ranges $0.02 \leq h/a \leq 1$, $0.025 \leq w/a \leq 0.25$ and $1.3 \leq d/a \leq 1.65$. Experiments by Dalmont et al. involving resonance analysis covering a subset of parameters was in agreement with the simulation.

Benade & Murday (1967) also measured the end correction for a disk placed over the end of a tube. Using resonance analysis, Benade & Murday found (using the same nomenclature as Dalmont et al.)

$$\delta_{\text{disk}} - \delta_{\text{circ}} = a[0.61(d/a)^{0.18}(a/h)^{0.39}]. \quad (2.31)$$

The results of Benade & Murday and Dalmont et al. are compared in Figure 2.6 for varying values of h/a . The curves in this figure are for $d/a = 1.375$ and $w/a = 0.1$. Note that Benade & Murday do not include the wall thickness w as a parameter in their formula. Significant disagreement exists between these results, particularly for large h (where the effect is small).

Dalmont et al. (2001) also provide a fit-formula for the end correction of a perforated disk poised over a cylindrical flange (perforated keypads are found on e.g. modern flutes).

$$\delta_{\text{perf}} - \delta_{\text{circ}} = \frac{\delta_{\text{disk}} - \delta_{\text{circ}}}{1 + 5(\delta_e/a)^{-1.35}(h/a)^{-0.2}}, \quad (2.32a)$$

where

$$\delta_e/a = [1.64a/q - 0.15a/d - 1.1 + ea/q^2], \quad (2.32b)$$

e is the thickness of the disk and q is the radius of the perforation.

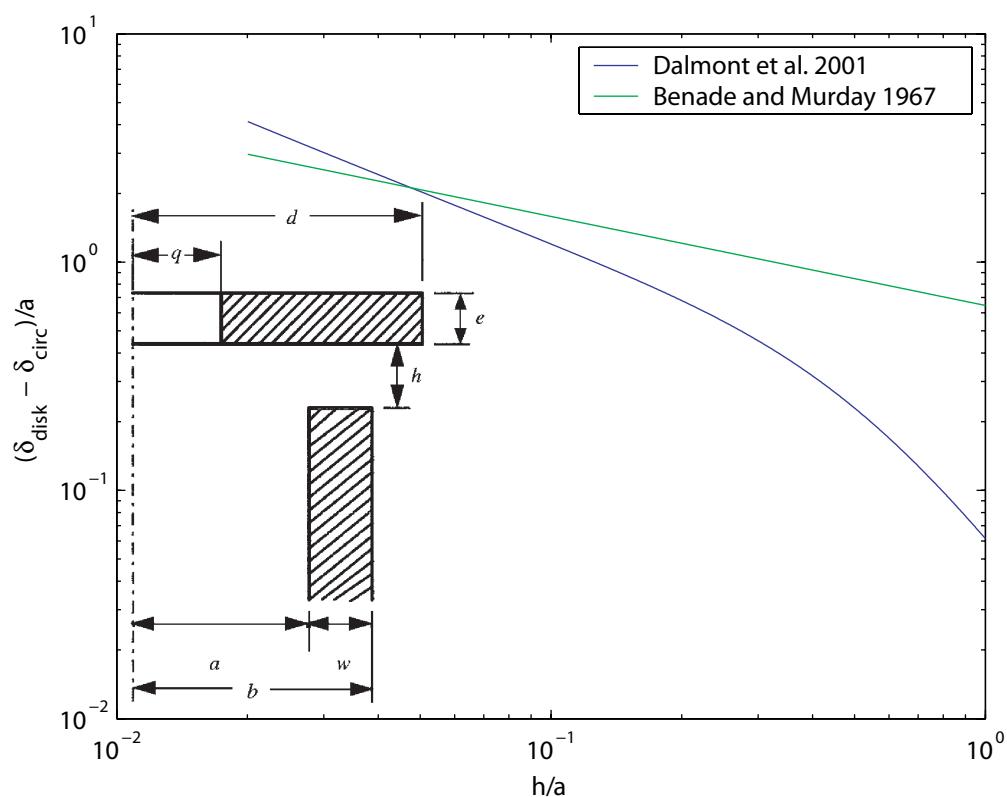


Figure 2.6: The increase in the end correction due to radiation for a disk poised over the open end of a cylindrical tube, as a function of disk height relative to tube radius. For the values of other parameters used, see text.

In most practical situations, the geometry of a keypad placed over a tone hole differs from the simplified geometries considered here, and some empirical corrections may be needed before applying these results to real tone holes. Coltman (1979) measured the reactances of real flute tone holes but did not separate the length corrections into inside and outside corrections.

2.2.9 Transmission line theory

Many of the methods and results of electrical transmission line theory can be applied to an analysis of woodwind instruments, with the acoustic pressure p taking the place of electric potential and the volume flow U replacing electric current. A small section of air within an instrument then has an inertance (analogous to electrical inductance) and a compliance (analogous to electrical capacitance). Since the wavelength in woodwind instruments is not large compared to the dimensions of the instrument, we must use transmission lines (or transfer matrices) to model the bore, rather than simple lumped elements. Lumped elements are useful, however, in analysing components that are small compared to the wavelength, such as tone holes.

In applying transmission line theory to a woodwind instrument, one must first represent the instrument bore as a succession of cylindrical and conical elements. This is exact in the limit as the length of the elements approaches zero.

Tone holes are represented in the model as T- or Π -circuits with impedance elements given by e.g. §2.2.10. The transfer matrix for an instrument relates the pressure and volume flow at the output to the same quantities at the input, and may be derived by multiplying together the transfer matrices of each element. The transfer matrices for cylindrical and conical pipe sections are given in §2.2.5 and §2.2.6. The transfer matrix for a tone hole with series and shunt impedances Z_a and Z_s is (Keefe 1990)

$$\mathbf{T} = \begin{bmatrix} 1 & Z_a \\ \frac{1}{Z_s} & 1 \end{bmatrix}, \quad (2.33)$$

provided $Z_a \ll Z_s$. I discuss Z_a and Z_s in the next section.

2.2.10 Tone hole equivalent circuits

At the junction between a tone hole and the bore, impedance corrections must be added to the one-dimensional model. In most cases, these can be added as length corrections (frequency-dependent in accurate calculations) to the hole segment and bore segments on each side. Figure 2.7 shows the geometry and dimensions of a simple cylindrical side hole. Dimensions a and b are the bore and hole radius respectively, and for convenience we shall use $\gamma = b/a$. Shown in Figure 2.8 is the substitution circuit for a symmetrical side hole between two tube sections. The impedance of the side hole itself is a summation of three impedances:

1. Z_h , the impedance of a tube with dimensions equal to the hole dimensions (as defined in Figure 2.7)
2. Z_r , the radiation impedance (see §2.2.8)
3. Z_m , the ‘matching volume’ correction.

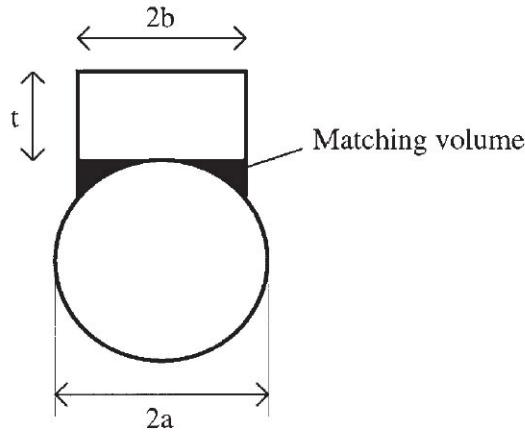


Figure 2.7: A cylindrical side hole, showing dimensions and the ‘matching volume’ (Dalmont et al. 2002).

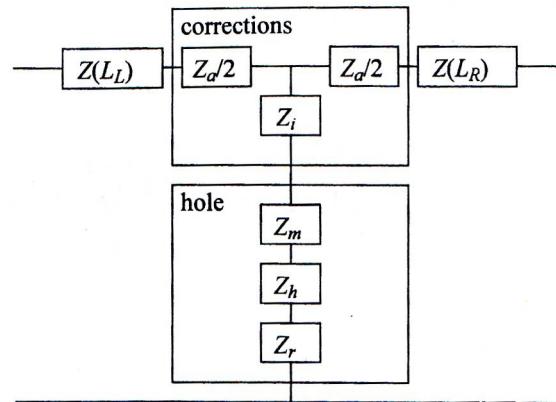


Figure 2.8: The equivalent circuit for a cylindrical side hole (Nederveen et al. 1998).

Z_m must be added to the impedance of the hole since the hole does not meet the main bore of the instrument at a plane. The ‘matching volume’ illustrated in Figure 2.7 may be accounted for by a length correction t_m added to the hole segment. A fitting formula for t_m was calculated by Keefe (1990) and modified by Nederveen et al. (1998) to make it accurate to within ± 0.001 :

$$t_m = \frac{b\gamma}{8}(1 + 0.207\gamma^3). \quad (2.34)$$

If the tone hole is slightly undercut, the correction t_m may be modified so that the equivalent volume includes the volume of the removed wall material.

The impedance corrections Z_a and Z_i shown in Figure 2.8 arise from higher mode effects in the vicinity of the hole. The series impedance Z_a is a negative inertance and arises from the flow entering the side hole to an extent (Figure 2.9a). The acoustic mass is thus reduced in the vicinity of the hole, and the effect is similar to a reduction in the main bore length, scaled by $\cos^2(kL_R)$, where L_R is the distance to the nearest flow antinode. Figure 2.9b shows the flow and potential lines for an open tone hole with all of the flow shunted by the hole. The shunt impedance is the impedance of the hole section (including the radiation impedance and

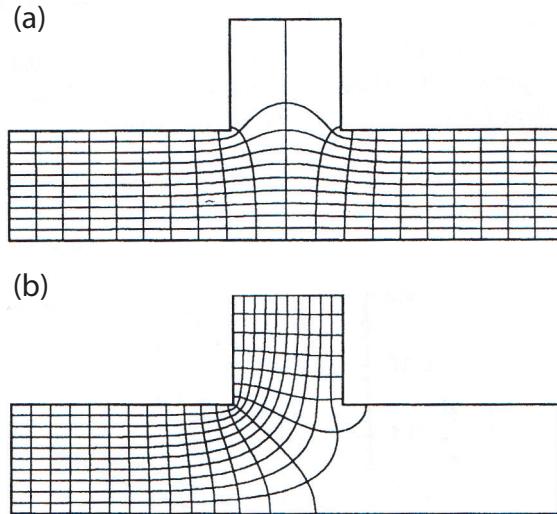


Figure 2.9: Flow and potential lines for a closed side hole at a pressure node (a) and for an open side hole and closed right hand tube piece (b) (Nederveen et al. 1998).

'matching volume' correction) plus the impedance correction Z_i . This 'inner' correction can be thought of as 'internal radiation' and in the limiting case of a very small hole is equal to the end correction for an infinite flange. If the side hole is long enough that the evanescent higher modes evoked at the inside and outside ends do not interact, then Z_a and Z_i are the same for open and closed holes. For very short holes (such as are found on e.g. the modern flute) Z_a will differ depending on whether the hole is open or closed, since the extent of flow widening may be reduced by the presence of the keypad. The substitution circuit shown in Figure 2.8 may also not strictly hold, due to interaction between the inside and outside fields.

Keefe (1982*b*) applied modal decomposition to the problem of a cylindrical side-branch in a cylindrical tube and gives formulae for the imaginary parts of the impedances in the T-circuit describing the hole. Keefe did not separate the components in the shunt impedance as shown in Figure 2.8, giving instead formulae for Z_s , the symmetrical or shunt impedance for the hole. Dubos et al. (1999) extended this work, correcting some errors and also treating the hole as a three-port. The results of Dubos et al. are more directly comparable to results of other authors. Nederveen et al. (1998) used the finite difference method to treat the same problem and derived fit-formulae similar to those of Dubos et al. Dalmont et al. (2002) compare the results of both theoretical papers and give some experimental results for the equivalent circuit of a side hole at low and high levels.

Recently, a hybrid method using the method of moments and finite difference methods has been used to analyse undercut tone holes in woodwinds (Poulton 2005).

2.2.10.1 Formulae for the series impedance

Dubos et al. (1999) give the following formulae for the series length correction for an open and a closed tone hole:

$$t_a^{(o)} = \frac{-b\gamma^2}{1.78 \tanh(1.84t/b) + 0.940 + 0.540\gamma + 0.285\gamma^2} \quad (2.35)$$

$$t_a^{(c)} = \frac{-b\gamma^2}{1.78 \coth(1.84t/b) + 0.940 + 0.540\gamma + 0.285\gamma^2}. \quad (2.36)$$

For long side holes ($t > b$) the series length corrections for open and closed side holes are approximately equal. The length correction is then (Dubos et al. 1999)

$$t_a = -(0.37 - 0.087\gamma)b\gamma^2. \quad (2.37)$$

Nederveen et al. (1998) give a slightly different formula for the series length correction:

$$t_a = -0.28b\gamma^2. \quad (2.38)$$

Equations (2.37) and (2.38) agree for $\gamma = 1$. The difference between the formulae is not particularly significant since the length correction becomes negligible for small holes.

2.2.10.2 Formulae for the inner impedance

The length correction t_i for the inner transitional impedance Z_i is given by Nederveen et al. as

$$t_i = (0.82 - 1.4\gamma^2 + 0.75\gamma^{2.7})b \quad (2.39)$$

and by Dubos et al. (see Dalmont et al. 2002) as

$$t_i = t_s - t_a\gamma^2/4, \quad (2.40)$$

where $t_s = (0.82 - 0.193\gamma - 1.09\gamma^2 + 1.27\gamma^3 - 0.71\gamma^4)b$, and the dependence on t_a is due to a different definition of the impedance components in the T-circuit.

Benade & Murday (1967) measured the inner length correction for an open hole of radius b drilled through the wall of a pipe with inside radius a . The inner correction E_i ($= (t_m + t_i)/b$ in the current nomenclature) is given by $E_i = [1.3 - 0.9(b/a)]$ for $(b/a) \leq 0.72$.

2.2.11 The open hole cutoff frequency

In a simple application of transmission line theory to woodwind musical instruments, it is very useful to consider an array of open tone holes as somewhat like a high-pass filter. This is because at sufficiently high frequencies the mass of air in an open hole is too great to accelerate efficiently, and to an acoustic wave the open hole presents an impedance much greater than that of the downstream section of the instrument. Benade (1976) derives a theoretical expression for the cutoff frequency of a continuous waveguide approximating this situation. For an array of open tone holes spaced uniformly at a distance $2s$ from each other, where the holes all have radius b and the bore has radius a , the cutoff frequency is given by

$$f_{\text{cutoff}} \approx 0.11 \frac{b}{a} \frac{c}{\sqrt{t_e s}}, \quad (2.41)$$

where t_e is the effective length of the hole (i.e. including end corrections). Wolfe & Smith (2003) derive this formula for an infinite array of tone holes.

The cutoff frequency increases with hole size and is one of the main reasons for the difference in tone between baroque, classical and modern flutes. Flutes with larger holes have a higher cutoff frequency and so produce sounds with a wider spectral content. These instruments sound ‘bright’ in comparison to smaller-holed instruments. The cutoff frequency also contributes to the highest note playable on the instrument. For most woodwind instruments the cutoff frequency is in the range 1.5–2.0 kHz. For a detailed discussion of the influence of the cutoff frequency on various types of flutes, see Wolfe & Smith (2003). In Chapter 5 impedance spectra of flutes and clarinets are compared with regard to their cutoff frequency.

2.2.12 Air-jet–resonator interactions

A flutist plays a flute by directing a jet of air across the embouchure hole. In the steady-state, i.e. when a standing wave is already established in the flute, the sound field within the flute imposes oscillations on the jet of air, alternately directing the air in and out of the flute. If the travel time of oscillations on the jet from the player’s embouchure to the embouchure edge is such that energy is added to the standing wave in phase, then the standing wave is sustained. For this reason the jet length and speed are critical, and a player can play several notes with a single fingering simply by changing the blowing pressure or the position of the lips. The jet is driven by acoustic flow into and out of the embouchure hole—therefore flutes operate near impedance minima, in contrast to reed instruments such as the clarinet and bassoon, which operate near impedance maxima.

The jet drive mechanism in flutelike instruments is inherently a nonlinear process, due in large part to the shape of the velocity profile of the jet. As discussed by Fletcher & Rossing (1998, chapter 16) the velocity profile of a jet has a bell-shaped form in the direction parallel to the axis of the embouchure hole. Due to this non-uniform velocity profile, different harmonic components are generated depending on the offset of the jet relative to the sound-generating edge. For example, a symmetric jet generates no even harmonics and the third harmonic can be made to vanish by inclining the jet into the instrument (in this case the first and second harmonics are still strong). The harmonic content of the jet drive also depends on the amplitude of excitation, with (in general) higher harmonic content at higher amplitude. Given such a nonlinear sound production mechanism, the playing frequencies of flutes are expected to vary with the level of excitation, and with jet offset. The playing frequency of a note may depend on the frequencies of more than one input impedance minima, and if these are not harmonically related, then the relative weighting will vary with amplitude.

A detailed discussion of the physics of air jets is beyond the scope of this work. For an overview see Fletcher & Rossing (1998) and for a review of physical models see Fabre & Hirschberg (2000). Experimental investigations in the context of the flute and organ pipes have been provided by Coltman (1968*a,b*), Fletcher & Thwaites (1979, 1983), Thwaites & Fletcher (1980, 1982) and Vergez et al. (2005).

2.2.13 The embouchure hole and stopper

For our purposes, it is important to note that flutes have a small volume of air between the embouchure hole and stopper. The stopper is usually set at 19.0 mm from the embouchure in the case of classical flutes and 17.0 mm for modern flutes but it may be adjusted to suit individual playing styles and intonations. The purpose of this upstream air volume (together with the bore taper) is to compensate for the frequency-dependent impedance at the far end of the flute, and thereby to bring the octaves into tune. Benade (1976) discusses the effect of the bore profile and the upstream volume, and Wolfe et al. (2003) show the effect of the stopper position on the input impedance of transverse flutes. The volume of air between the embouchure hole and stopper also limits the range of the flute, since at high frequencies the air acts as the compliant element in a Helmholtz oscillator (the mass of air in the embouchure hole riser acts as the inertance). At the resonant frequency of this oscillator (approx. 7 kHz, well above the playing range) the acoustic fields do not propagate substantially into the body of the flute and the flute does not respond to changes in fingerings.

2.3 COMPUTER MODELS

Plitnik & Strong (1979) applied transmission line theory to the oboe and calculated the input impedance for various fingerings. The authors compare the predicted impedance spectra with measurements made by A. H. Benade. Although the results may not be accurate enough for precise prediction of the tuning and playing qualities of oboes, the model provides a reasonably good fit to experiment, and allows validation of the assumptions made. Strong et al. (1985) used a numerical model to calculate impedances and standing wave patterns of the modern flute. The results are compared to experiment and discrepancies in resonance frequencies are discussed.

The numerical model of woodwind instruments developed by Keefe (1990) is very similar to that used in this work. Keefe applies the model to the air column of a flute and finds good agreement with experiment. However, no examples are given of the degree of accuracy of the model in predicting the tuning of particular notes on the flute.

Several attempts have been made to produce a stand-alone computer program to predict the playing qualities of woodwinds based on input geometrical parameters. Flutekey (Coltman 1998) is one such program, made by John Coltman. Flutekey is a MS-DOS based program that calculates the tuning of notes on the modern (Boehm) flute with different hole parameters. Resonans* is a software package developed jointly by Le Mans University and IRCAM (Institut de Recherche et Coordination Acoustique/Musique), Paris, allowing computer aided design of wind instruments. The particular details of this software package are not readily available, although the software is said to have been used successfully by several instrument makers.

The Virtual Flute (Botros et al. 2006) uses a transmission line model like that of Plitnik &

*Some information about Resonans may be found at <http://www.ircam.fr>.

Strong (1979) with a half-dozen empirical parameters to model the modern flute. These parameters are machine-learned from a training set of impedance spectra for all standard fingerings and some alternative fingerings. The model is accessed through a user-friendly web page. The software in its current form does not allow geometrical changes to the flute model, as it is intended primarily for use by flute players rather than makers.

This work extends the model used in The Virtual Flute, making it accurate for flutes of varying geometry. While the idea of a flute model itself is not novel, its close association with a highly accurate and extensive database of flute impedance spectra should provide the accuracy and reliability required by instrument makers. Furthermore, a great deal of effort was put into the design of the user interface to the model. The use of a simple graphical interface widens significantly the pool of potential users of the software, making the work of musical acousticians much more widely available.

Chapter III

Measuring acoustic impedance*

Resonances and/or singularities during measurement and calibration often limit the precision of acoustic impedance spectra. This chapter reviews and compares several established techniques, and describes a technique that incorporates three features that considerably improve precision. The first feature is to minimise problems due to resonances by calibrating the instrument using up to three different acoustic reference impedances that do not themselves exhibit resonances. The second involves using multiple pressure transducers to reduce the effects of measurement singularities. The third involves iteratively tailoring the spectrum of the stimulus signal to control the distribution of errors across the particular measured impedance spectrum. Examples are given of the performance of the technique on simple cylindrical waveguides.

3.1 INTRODUCTION

The input impedance of any one-dimensional waveguide is defined as the complex ratio of pressure to volume flow at the input. This quantity is used to describe the linear acoustics of automotive mufflers, air-conditioning ducts and the passive elements of wind instruments and the vocal tract. For a musical instrument, the input impedance usefully displays important characteristics of the instrument in the absence of a player, and indicates how the instrument will respond when excited at any frequency. In this case, high resolution in magnitude and frequency are particularly important. If pressure and volume flow are measured at different points in a system, their ratio gives a transfer impedance, which is particularly useful in characterising multi-port systems.

In this paper, we review the various approaches to measuring acoustic impedance and calibrating impedance heads and propose a general calibration technique for heads with multiple transducers. We consider the effect of transducer errors on impedance measurements and present a technique for distributing any measurement errors over the frequency range. To demonstrate the technique we use an impedance head with three microphones to measure the input impedance of simple cylindrical waveguides. The effects of calibration and optimisation on these measurements are presented and discussed.

*An abridged version of this chapter has been published as Dickens, P., Smith, J. & Wolfe, J. (2007), 'Improved precision in measurements of acoustic impedance spectra using resonance-free calibration loads and controlled error distribution', *J. Acoust. Soc. Am.* **121**(3), 1471–81.

3.2 REVIEW OF MEASUREMENT TECHNIQUES

Many techniques for measuring acoustic impedance have been devised. The major techniques are reviewed by Benade & Ibisi (1987) and Dalmont (2001a). Any two transducers with responses that are linear functions of pressure and flow may be used to construct an impedance head; hence many designs are possible. In Table 3.1, several common techniques are illustrated, along with conditions for singularities. At a singularity, the system of equations governing an impedance head becomes degenerate, and the impedance cannot be determined (see §3.5.5).

In methods (a) and (b) (Table 3.1), a single pressure transducer (microphone) is used. In the volume flow source method (a) the input impedance is proportional to the ratio of the pressure measured with the unknown load to that with a reference load using the same stimulus. An attenuator ensures a source of volume flow, provided the impedance of the attenuator is much greater than that of the unknown load. (The effect of a finite source impedance can be reduced, to first order, by subtracting the attenuator admittance from the measured admittance.) In (b), a method known as pulse reflectometry, a pressure pulse is recorded as it travels toward the load, and again as it returns, yielding the impulse response function. This is mainly used for area reconstruction of musical wind instruments (Watson & Bowsher 1988) and the human airway (Fredberg et al. 1980) however the acoustic impedance can be obtained from the impulse response function after a Fourier transform.

In methods (c) to (e) two similar transducers are used simultaneously. In methods (c) and (e), neither transducer measures pressure or flow at the input to the load alone; these are obtained by computations involving the transfer functions of the duct and the transducer properties. If a linear attenuator is present between the two microphones, as in method (d), the pressure difference between the two microphones is proportional to the flow. Alternatively, a signal approximately proportional to flow may be obtained by measuring the pressure in a fixed cavity at the back of the driver (Singh & Schary 1978). Impedance heads have also been devised using a pressure transducer and a flow transducer (f), yielding the impedance with a minimum of computation (provided both transducers are close enough to the reference plane). Because of the difficulties in measuring flow precisely, these have limited dynamic range and signal-to-noise ratio.

The signal-to-noise ratio and frequency range may both be increased if an array of more than two transducers is used. Such a system with three microphones is shown in method (g).

3.3 THEORY OF ACOUSTIC IMPEDANCE MEASUREMENTS

A general impedance head is shown in Figure 3.1. Some source of acoustic energy (shown here as a loudspeaker) excites the air inside a conduit (usually cylindrical) and transducers along the conduit measure some signal proportional to pressure and flow. The impedance is measured at some reference plane, to which load impedances may be attached.

The air inside the conduit is excited at frequencies below the cut-on frequency of the first higher mode, which for a cylindrical duct of radius a occurs at $f = 1.84c/2\pi a$, where c is the

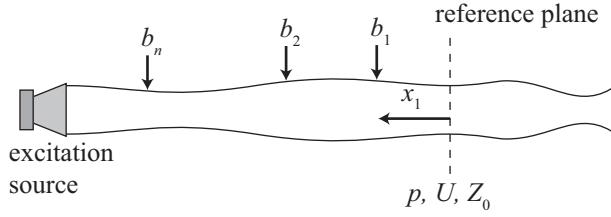


Figure 3.1: A generalised impedance head. Impedance is measured at the *reference plane*, where the pressure, volume flow and characteristic impedance are given by p , U and Z_0 respectively. The volume flow is positive flowing *into* the unknown load. n sensors record the signals b_1, b_2, \dots, b_n and are at positions x_1, x_2, \dots, x_n , measured from the reference plane.

speed of sound (Fletcher & Rossing 1998). Therefore, all modes except the plane wave mode are non-propagating. If higher modes are excited near the transducers and the reference plane, the measured impedance will be something other than the plane-wave impedance. The effect of discontinuities at the transducers may be removed by calibration. If a discontinuity exists at the reference plane, non-propagating modes are excited, introducing errors into the measured plane wave impedance. Hence it is preferable that the object under study couple smoothly to the measurement conduit—otherwise multimodal theory can be used to calculate a correction (see §3.5.3).

It should be noted that in some impedance heads, the source and any transducers are located at or very near the reference plane; the sketch in Figure 3.1 should be considered a general case only. However, it is often desirable to separate the source from the transducers, to ensure that the acoustic waves are planar at the transducers, subject to the constraints discussed in §3.5.

For an impedance head with n transducers ($n \geq 2$), the pressure p and flow U at the reference plane are given by the vector \mathbf{x} in a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ \vdots & \vdots \\ A_{n1} & A_{n2} \end{bmatrix} \begin{bmatrix} p \\ Z_0 U \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad (3.1)$$

where \mathbf{b} is a vector of transducer signals and the elements of each matrix are, in general, functions of frequency. In order for the matrix \mathbf{A} to be dimensionless, any quantities with units of volume flow are parameterised by the characteristic impedance of the head at the input, Z_0 . Thus \mathbf{x} and \mathbf{b} both have units of pressure.

For ideal transducers positioned a known distance from the reference plane and mounted in a cylindrical duct, the elements of \mathbf{A} are given by the transfer matrix for a straight tube. The signal from an ideal pressure transducer with unity gain at position x is given by

$$b_{\text{pressure}}(x) = \cosh(ikx)p + \sinh(ikx)Z_0U \quad (3.2)$$

and the signal ($b_{\text{flow}}(x) \equiv Z_0U(x)$) from an ideal flow transducer (again at x and with unity gain) is given by

$$b_{\text{flow}}(x) = \sinh(ikx)p + \cosh(ikx)Z_0U, \quad (3.3)$$

Table 3.1: Several of the more common impedance heads, with conditions for singularities, selected references and notes. In the expressions for singularities, k is the wavenumber and $n = 1, 2, 3, \dots$. The reference plane is indicated by a vertical dashed line.

Impedance Head	Singularities	Refs.	Notes
(a) volume flow source	$kd = (2n - 1)\frac{\pi}{2}$	Singh & Schary (1978), Benade & Ibisi (1987), Bruneau (1987), Wolfe et al. (2001a)	<ul style="list-style-type: none"> computationally simple requires calibration of the source prone to errors at high Z
(b) pulse reflectometer	f -range limited by pulse width	Fredberg et al. (1980), Watson & Bowsher (1988)	<ul style="list-style-type: none"> uses same microphone for incident and reflected wave; calibration unnecessary accuracy limited by length of measurement duct
(c) two microphones	$kd = (n - 1)\pi$	Seybert & Ross (1977), Chung & Blaser (1980a,b,c), Bodén & Abom (1986), Abom & Bodén (1988), Gibiat & Laloë (1990), van Walstijn et al. (2005)	<ul style="list-style-type: none"> fewer simplifying assumptions required computationally intensive
(d) volume flow source with upstream microphone	$kd = (2n - 1)\frac{\pi}{2}$	Backus (1974, 1976), Caussé et al. (1984)	<ul style="list-style-type: none"> signal from upstream microphone proportional to flow (for an attenuator with high impedance compared to the unknown load)

Continued on next page

Impedance Head	Singularities	Refs.	Notes
(e) two anemometers	$k d = (n - 1)\pi$	van der Eerden et al. (1998)	<ul style="list-style-type: none"> computationally similar to (c) several particle velocity sensors can be made simply with similar characteristics
(f) microphone and anemometer	$k d = (2n - 1)\frac{\pi}{2}$	Pratt et al. (1977), Elliott et al. (1982), Kob & Neuschafer-Rube (2002)	<ul style="list-style-type: none"> direct measurement of both pressure and flow correction required to obtain volume flow from particle velocity
(g) multiple microphones	vary with microphone spacings	Jang & Ih (1998)	<ul style="list-style-type: none"> wide frequency range increased precision

Legend:  sound source;  attenuator;  microphone;  flow sensor

where $k = \frac{\omega}{v} - i\alpha$ where $i = \sqrt{-1}$ and v and α (the phase velocity and attenuation coefficient) are calculated taking into account viscothermal loss (see e.g. Fletcher & Rossing 1998). From these, the matrix \mathbf{A} may be built up for any combination of transducers.

Once \mathbf{A} is determined, the pressure and flow (and hence the impedance) for a given measurement \mathbf{b} are obtained by solving (3.1). The equation is solved in the normal algebraic sense for $n = 2$. For $n > 2$, there are more equations than are algebraically necessary to determine the pressure and flow. In this case (3.1) is solved using a least-squares method.

Determining \mathbf{A} from theory does not take into account perturbation of the wave by the transducers or non-identical transducer responses and also requires an accurate knowledge of the complex wavenumber k , which depends significantly on temperature, humidity and surface roughness. For these reasons, one or more calibrations are often used to determine \mathbf{A} .

3.4 CALIBRATION OF IMPEDANCE HEADS

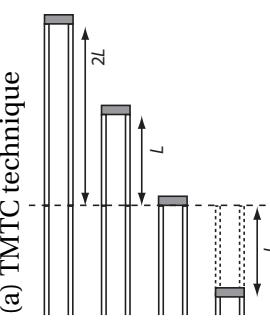
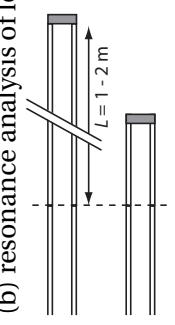
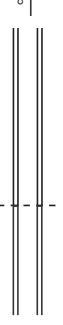
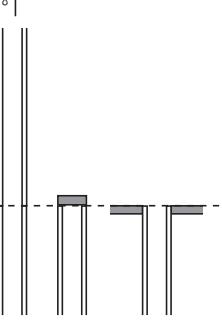
3.4.1 Review of calibration techniques

An impedance head can be constructed with two or more transducers that respond linearly to changes in pressure and flow. This includes, for example, microphones with non-negligible compliance and transducers with frequency-dependent gain. These are mounted in a waveguide which need not be cylindrical, and the exact positions of each transducer need not be known. Such a system may be calibrated fully by measuring its response to three test loads of known impedance. For a head with two microphones, three complex parameters must be determined: e.g. the gain ratio of the microphones, the admittance of the microphone closest to the unknown load, and the complex wavenumber for propagation within the waveguide, although other physical quantities such as the distance to each microphone may be used as parameters. Hence the ratios of microphone signals from three known loads are required to calibrate the system fully. If there are more than two microphones, the calibration parameters are overdetermined with three calibration loads, but two is not enough. If some assumptions can be made about the impedance head, then the number of calibrations required is reduced.

Gibiat & Laloë (1990) give a complete calibration routine for the two-microphone method (calibration loads (a) in Table 3.2). They use two stopped pipes of diameter equal to that of the measurement head, and a quasi-infinite impedance (a solid stop at the reference plane) to determine three calibration functions. The lengths of the two stopped pipes must be chosen carefully so that a range of impedances is encountered at each frequency. For example, if a calibration pipe has a resonance at a frequency of interest, its input impedance at that frequency will be very similar to that of the solid stop, and the calibration functions will have large errors at that frequency. For measurements of acoustic impedance over a wide frequency range, several microphone spacings are needed, each with its own set of calibration loads.

The two-microphone-three-calibration (TMTC) technique described earlier (Gibiat & Laloë 1990) depends on accurate knowledge of the impedances of the test pipes; this in turn requires accurate knowledge of the complex wavenumber, a quantity that is strongly dependent on measurement conditions and the surface characteristics of the test pipes. Calibration using

Table 3.2: Several techniques for calibration of impedance heads, with selected references and notes.

Calibration Loads	Refs.	Notes
(a) TMTC technique	Gibiat & Lalöe (1990), van Walstijn et al. (2005)	<ul style="list-style-type: none"> 'complete' calibration with three known loads several sets of calibration loads needed to cover wide f-range complex wavenumber need not be known if a fourth 'negative' length load is used 
(b) resonance analysis of long tube	Bruneau (1987), Dalmont (2001 <i>b</i>)	<ul style="list-style-type: none"> after initial measurement of Z for long tube, calibration parameters determined from oscillations in a and k complex wavenumber need not be known if an extra short tube is used data obtained only at resonances and antiresonances of long tube—low frequency limit determined by length of tube 
(c) semi-infinite pipe	Wolfe et al. (2001 <i>a</i>)	<ul style="list-style-type: none"> almost purely resistive load—impedance insensitive to complex wavenumber used for calibration of volume flow sources 
(d) resonance-free loads		<ul style="list-style-type: none"> complete calibration as in (a) valid for all frequencies, due to lack of resonances the flange calibration may be omitted if a model of the impedance head is available 

resonant pipes explicitly depends on a theory for wall losses. Further, the temperature and humidity must be accurately determined, and the test pipes must be very accurately machined. If an extra calibration is available, then the complex wavenumber need not be known; this is the approach of van Walstijn et al. (2005), who employ the three calibrations of Gibiat & Laloë plus a ‘negative length’ tube, realised by defining the reference plane some distance from the first microphone.

Dalmont (2001*b*) presents a calibration technique for impedance heads with a volume flow source, which may also be extended to the two-microphone case (Dalmont 2001*a*) (calibration loads (b) in Table 3.2). The method is based on resonance analysis of a long closed tube. The impedance of the tube is measured using the uncalibrated head and the attenuation coefficient and wavenumber are derived from the measurement. When plotted against frequency these will show periodic oscillations with amplitude proportional to any errors in calibration. The three calibration constants are estimated from these oscillations, and the procedure is repeated until the oscillations are tolerably small. The main advantage of this procedure is that it does not depend on exact knowledge of the complex wavenumber. The main disadvantage is that it only yields calibration data at each resonance frequency of the calibration tube. These may be interpolated, but the lowest frequency that can be measured is limited by the length of the calibration tube (and is around 80 Hz for a 2 m tube—and higher for shorter tubes).

3.4.2 General calibration technique using up to three resonant-free loads

Here, we calibrate a three-microphone impedance head in a method similar to that used by Gibiat & Laloë. However, to obviate the need to know the complex wavenumber precisely, we use three loads without any resonances: a quasi-infinite impedance; an almost purely resistive impedance; and a flange (calibration loads (d) in Table 3.2). The resistive impedance is in our case a pipe so long that the reflected wave returns reduced in amplitude by 80 dB or more. For lower frequencies, with loss less than 80 dB, it suffices to deliver the signal in pulses of duration T with $T < 2L/c$, where L is the length of the pipe.

The three calibration loads (∞ , Z' and Z'') are measured and yield the measurement vectors \mathbf{b} , \mathbf{b}' and \mathbf{b}'' . The measurements are made and the output spectrum optimised (see §3.6) using either a theoretical matrix \mathbf{A} derived from (3.2) or one derived from a previous calibration on the same measurement head. We then have the three calibration equations

$$p\mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \mathbf{b} \quad (3.4)$$

$$p'\mathbf{A} \begin{bmatrix} 1 \\ 1/\bar{Z}' \end{bmatrix} = \mathbf{b}' \quad (3.5)$$

$$p''\mathbf{A} \begin{bmatrix} 1 \\ 1/\bar{Z}'' \end{bmatrix} = \mathbf{b}'' \quad (3.6)$$

where the bar represents a reduced impedance ($\bar{Z} \equiv \frac{Z}{Z_0}$) and p , p' and p'' are the pressures at the reference plane during measurement of each of the three calibration loads. Dividing each

subsequent row in (3.4) by the first row yields the first column of \mathbf{A} ,

$$A_{j1} = A_{11}b_j/b_1, \quad (3.7)$$

in terms of A_{11} . A_{11} can be given any value without affecting impedance measurements; it is usually set equal to $\cosh(ikx_1)$ (equivalent to assuming the first microphone has unity gain and that the measurement duct is cylindrical).

Taking pairs of rows from (3.5), we may eliminate p' and obtain linear equations in the unknowns A_{j2} . For example, rows 1 and 2 combine to give $b'_2(A_{11} + A_{12}/\bar{Z}') = b'_1(A_{21} + A_{22}/\bar{Z}')$. The elements A_{j2} are determined by eliminating the pressure (p' or p'') from each pair of rows in (3.5) and (3.6) and solving the resulting system. Note that for all $n > 2$ the system is overdetermined (in the algebraic, noise-free sense) and for $n = 2$ is algebraically equivalent to the TMTC technique of Gibiat & Laloë.

So, e.g. for a head with two microphones, calibrated with three known loads,

$$\begin{bmatrix} b'_2/\bar{Z}' & -b'_1/\bar{Z}' \\ b''_2/\bar{Z}'' & -b''_1/\bar{Z}'' \end{bmatrix} \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} = \begin{bmatrix} b'_1 A_{21} - b'_2 A_{11} \\ b''_1 A_{21} - b''_2 A_{11} \end{bmatrix}. \quad (3.8)$$

The above-outlined calibration technique assumes very little about the geometry of the impedance head and the characteristics of the transducers. If the calibration is complete, then wall losses within the impedance head do not need to be taken into account explicitly. In the multiple microphone technique with cylindrical waveguide and microphones attached at known distances from the reference plane, the calibration parameters may be recast in a more instructive form. If each microphone has an admittance of y_j/Z_0 , then the pressure and upstream flow at microphone j are related to those at microphone $j-1$ according to

$$\begin{bmatrix} p_j \\ Z_0 U_j^+ \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -y_j & 1 \end{bmatrix} \mathbf{T} \begin{bmatrix} p_{j-1} \\ Z_0 U_{j-1}^+ \end{bmatrix}, \quad (3.9a)$$

where \mathbf{T} is the transfer matrix for a cylindrical pipe

$$\mathbf{T} = \begin{bmatrix} \cosh(ikd_j) & \sinh(ikd_j) \\ \sinh(ikd_j) & \cosh(ikd_j) \end{bmatrix} \quad (3.9b)$$

and $d_j = x_j - x_{j-1}$. For $j = 1$, p_{j-1} and U_{j-1}^+ are the pressure and flow at the reference plane and $d_1 = x_1$. The microphone signals are equal to $\kappa_j p_j$, where κ_j is the gain of microphone j . Taking (3.9) and a calibrated matrix \mathbf{A} , one can determine k , κ_j for $j = 1, \dots, n$ and y_j for $j = 1, \dots, n-1$ (y_n , the dimensionless admittance of the microphone closest to the source, cannot be determined). For a measurement set-up with a combination of pressure and flow transducers, or all flow transducers, the calibration proceeds in much the same way and an equation similar to (3.9) can be constructed.

Sometimes a third calibration is unnecessary or impracticable. In these cases one may pre-calculate the complex wavenumber k , using a theory that accounts for viscothermal losses within the waveguide. For a given impedance head, a single set of three calibrations can determine the degree of confidence one may take in this assumption, and the errors involved in making it. The remaining elements of \mathbf{A} are then found from (3.4) and (3.5) as described.

If one were confident in making further assumptions about the impedance head, then a single calibration may be used to determine either the microphone gains or admittances. For example, one might assume that $y_j = 0$ for $j = 1, \dots, n - 1$ (a reasonable assumption for small microphones coupled closely to a large waveguide). Then the matrix elements A_{j1} would be found from a measurement of the quasi-infinite impedance load (3.7) and the elements A_{j2} are given by

$$A_{j2} = A_{j1} \tanh(ikx_j). \quad (3.10)$$

Otherwise, we may decide to assume that $\kappa_j = 1$ for all j and determine the microphone admittances from the quasi-infinite impedance calibration using (3.9).

3.4.3 Choice of calibration loads

The TMTC technique works well for small frequency ranges but depends critically on the theory used to account for wall losses. Larger frequency ranges can be covered by using several microphone spacings and calibration tubes. By using the general technique presented earlier, where the signals from two or more microphones are processed simultaneously, a wide frequency range can be covered without measuring piece-wise. The three resonant-free calibration loads used here are sufficient to determine the calibration parameters over the entire frequency range, although the impedance for the flange calibration must be derived from theory. For an impedance head with cylindrical waveguide, and nearly ideal microphones at known distances from the reference plane, one or more of the calibration loads may be omitted. Thus calibrating with the quasi-infinite impedance alone may be sufficient for many applications. If another calibration is required and a resistive impedance load is not available, one or more closed tubes of different length may be used instead. Then \bar{Z}' in (3.5) is the reduced impedance of the closed tube and several such equations should be solved simultaneously if several closed tubes are used (to choose resonant tube lengths see Gibiat & Laloë 1990).

3.5 ERRORS

3.5.1 Inadequate spectral resolution

Large resonances are usually present in any duct system used to measure acoustic impedance. As discussed by Bodén & Åbom (1986) in the context of the two-microphone method, the pressure spectrum at each microphone varies periodically with frequency, due both to changes in the standing wave pattern in the duct as frequency is varied, and to resonances of the total duct system. When this pressure spectrum is estimated with a frequency resolution Δf , the ‘period’ (in the frequency domain) of any variation in the spectrum must be large compared to Δf in order to avoid errors associated with resonances in the head and impedance system. For this reason, the microphones should be positioned as close as possible to the impedance to be measured, and the duct length should be kept small. Some acoustic damping between the loudspeaker and the transducers may be used to reduce the amplitude of these duct resonances.

3.5.2 Non-linear transducer responses

Microphone and loudspeaker distortion can both produce errors in a measured impedance. In methods where the impedance is calculated solely from the signals of two or more transducers obtained simultaneously, a distorting loudspeaker will not affect the measurement, as the distortion is present in each transducer signal and is cancelled out. In methods using a single transducer and attenuator, loudspeaker distortion will only affect the measurement if it changes with the load (possible at impedance extrema). Distortion in the transducers will produce errors in the measured impedance. Microphone distortion is always present but is reduced at lower pressures, and the excitation signal may be adjusted to achieve a compromise between random noise and distortion. In this study, the compromise between these two effects was made during calibration on the quasi-infinite impedance by adjusting the output signal level to give microphone signal ratios with minimal contamination from random noise or distortion (as measured by the deviation of the signal from a smooth curve on a small frequency scale).

3.5.3 Diameter mismatch at the reference plane

At any bore discontinuity, non-propagating modes are evoked. Automotive mufflers usually have several such discontinuities. Most woodwind instruments have such discontinuities (for example at tone holes), but provided they are located a sufficient distance from the input, they do not influence the measurement of the plane-wave impedance. If, however, a discontinuity is present at the reference plane, the measured impedance will not be the plane-wave impedance of the object but some combination of the elements of the generalised impedance matrix described by Pagneux et al. (1996).

The difference between the measured impedance and the ‘true’ plane-wave impedance may be determined by multi-modal theory and expressed as an error term. Alternatively, if an impedance head is calibrated on pipes with entry diameter equal to the entry diameter of the object under study, higher modes evoked at the reference plane are automatically taken into account (Gibiat & Laloë 1990). In practice this approach requires many sets of calibration pipes, and it is often easier to apply one of the following corrections.

Van Walstijn et al. (2005) discuss the effect of a duct discontinuity in the context of the two-microphone technique, but the results are applicable to any system where the transducers are a sufficient distance from the reference plane to measure only plane waves. They show that the true impedance is related to the measured impedance according to

$$Z_{\text{true}} \approx Z_{\text{meas}} - \sum_{n=1}^N (F_{0,n})^2 Z_{n,n}^{(c)}, \quad (3.11)$$

where Z_{true} and Z_{meas} are, respectively, the plane wave impedances on the object and measurement sides of the discontinuity. $F_{0,n}$ is given by Pagneux et al. (1996, equation (56)), and $Z_{n,n}^{(c)}$ are the diagonal elements of the characteristic impedance matrix $\mathbf{Z}^{(c)}$:

$$Z_{n,n}^{(c)} = \frac{\omega\rho}{k_n S}, \quad (3.12)$$

where k_n is the propagation wave vector for the n th axisymmetric mode (see Fletcher & Rossing 1998, equation (8.7)), S is the entry cross-sectional area of the object and ω and ρ are the

angular frequency and the air density. This treatment assumes that all higher modes excited at the reference plane are evanescent and do not couple to any higher modes in other parts of the object.

Incidentally, we may rewrite (3.11) in terms of the pressure and flow. The plane-wave flow is continuous at a duct discontinuity ($U_{\text{true}} = U_{\text{meas}} = U$) and so the true plane wave pressure is given by

$$p_{\text{true}} \approx p_{\text{meas}} - U \sum_{n=1}^N (F_{0,n})^2 Z_{n,n}^{(c)}. \quad (3.13)$$

For the case of an axially symmetric impedance head comprised of an acoustic resistance of annular cross section and a microphone, both located at the reference plane, Fletcher et al. (2005) provide an equation similar to (3.11):

$$Z_{\text{true}} \approx Z_{\text{meas}} - \sum_{n=1}^N \frac{\omega \rho}{2\pi k_n M_n} J_0\left(\frac{\alpha_n a}{R}\right), \quad (3.14)$$

where a and R are, respectively, the radius of the annulus and the entry radius of the object under study, and α_n is the n th zero of the first-order Bessel function of the first kind. The factor M_n is given by

$$M_n = \frac{R^2}{2} J_0^2(\alpha_n). \quad (3.15)$$

In both of these cases, the number of higher modes included in the sum, N , should be chosen so that the sum converges. Van Walstijn et al. used 40 and Fletcher et al. 100 higher modes to be certain of achieving a good approximation, although this will of course depend on the size of the diameter mismatch.

Brass & Locke (1997) discuss the effect of the evanescent wave on impedance measurements of the ear canal with a loudspeaker and microphone in close proximity.

An impedance determined by applying the above-mentioned corrections will usually be less accurate than one measured with a matching impedance head, as turbulent losses are not taken into account in the multi-modal model.

3.5.4 Random noise (acoustical or electrical)

Each transducer signal is contaminated by random noise. Whether this is of acoustical or electrical origin is usually unimportant. This noise often has an overall $1/f^n$ -dependance, where $0 < n < 1$ and the quality of measurements may be improved by increasing the power in the lower frequencies.

3.5.5 The ‘singularity factor’

The sensitivity of any impedance head to errors in the input quantities varies over frequency. In the two-microphone method, for example, the head becomes ‘singular’ when the microphone spacing is an integral multiple of $\lambda/2$ and in this vicinity large errors in impedance result from small measurement errors. Conversely, for a microphone spacing of $\lambda/4$, the head is least sensitive to errors in the measured quantities. This effect is conveniently represented by the function SF (for ‘singularity factor’), defined by Jang & Ih (1998, their equation (16)), and

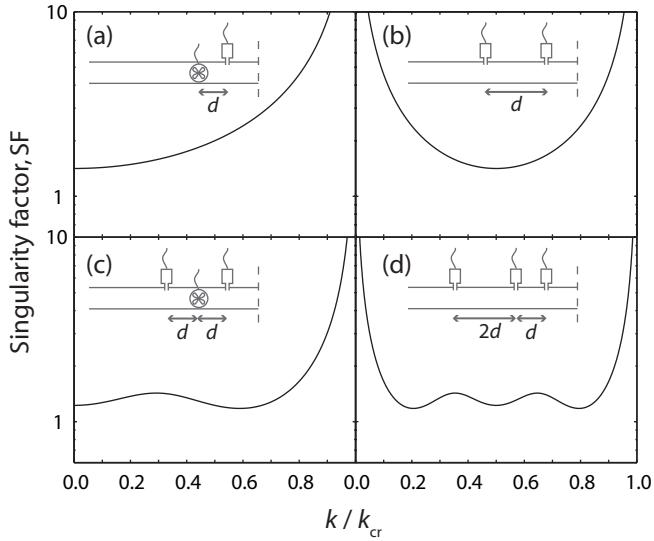


Figure 3.2: Singularity factors plotted from $k = 0$ to the first singularity for several impedance heads (shown schematically in inset). The heads comprise (a) a microphone and anemometer, (b) two microphones, (c) two microphones and an anemometer and (d) three microphones. For (a) and (c) $k_{cr}d = \frac{\pi}{2}$ while for (b) and (d) $k_{cr}d = \pi$.

derived from the singular value decomposition of the matrix \mathbf{A} . (Jang & Ih are primarily interested in reflection coefficients, and their matrix \mathbf{A} relates the incident and reflected waveforms to the measured pressures. Their expression for SF remains valid when our modified matrix \mathbf{A} is used.) The singularity factor is useful to compare different impedance heads according to their sensitivity to measurement errors. SF is plotted for four systems in Figure 3.2. Here attenuation is neglected and ideal transducers are assumed. The head in (a) utilises a flow transducer and a microphone coupled to the duct at different positions. SF for this set-up is smallest at low wavenumber and has a singularity when the distance between the transducers is equal to a quarter wavelength. In order to reduce error, the transducers should be as close as practicable to each other. However they are often separated by some distance, and the effect of this on the error in Z should not be neglected. Also, if the transducers deviate at all from ideality, they may be considered as ideal transducers separated by a certain effective ‘distance’ (which may be complex and frequency-dependent) (Dalmont 2001*b*), and two such transducers may exhibit behaviour as depicted in Figure 3.2 (a) even if physically located at the same position.

In Figure 3.2 (b) and (d) SF is shown for heads comprising two and three microphones, respectively. These functions are singular at $k = 0$ and when the smallest microphone separation is equal to $\lambda/2$. Addition of the third microphone reduces SF over the entire frequency range and widens the range over which the method may be used. In Figure 3.2 (c) SF is shown for two microphones and a flow transducer; not surprisingly the function is lower than that for either Figure 3.2 (a) or (b) in isolation.

3.5.6 Calculating the error in Z

Following Jang & Ih (1998), we may write the measured signals $\tilde{\mathbf{b}}$ as the sum of the (hypothetical) real values, \mathbf{b} , and the measurement errors \mathbf{m} :

$$\tilde{\mathbf{b}} = \mathbf{b} + \mathbf{m}. \quad (3.16)$$

The errors in \mathbf{x} may then be written in terms of the measurement errors:

$$\tilde{\mathbf{x}} - \mathbf{x} = \mathbf{A}^+ \mathbf{m} \quad (3.17)$$

where \mathbf{A}^+ is the generalised (Moore-Penrose) inverse of \mathbf{A} .

If we right-multiply the error in \mathbf{x} by its complex conjugate transpose (denoted here by a superscript H), the expectation value of the resulting matrix is given by

$$\begin{aligned} E[(\tilde{\mathbf{x}} - \mathbf{x})(\tilde{\mathbf{x}} - \mathbf{x})^H] &= E[(\mathbf{A}^+ \mathbf{m})(\mathbf{A}^+ \mathbf{m})^H] \\ &= E[\mathbf{A}^+ \mathbf{m} \mathbf{m}^H (\mathbf{A}^+)^H] \\ &= \mathbf{A}^+ \mathbf{V} (\mathbf{A}^+)^H \end{aligned} \quad (3.18)$$

where $\mathbf{V} = E[\mathbf{m} \mathbf{m}^H]$ is the covariance matrix. If each transducer signal is contaminated by uncorrelated noise of variance σ_j^2 , where σ_j can in general be a function of frequency, then \mathbf{V} is a diagonal matrix with the σ_j^2 's on the diagonal. The errors in p and U , Δp and ΔU are given by the diagonal elements of the matrix in (3.18):

$$|\Delta p|^2 = [\mathbf{A}^+ \mathbf{V} (\mathbf{A}^+)^H]_{11} \quad (3.19)$$

$$|Z_0 \Delta U|^2 = [\mathbf{A}^+ \mathbf{V} (\mathbf{A}^+)^H]_{22} \quad (3.20)$$

and the error ΔZ in the impedance Z is obtained by propagation of errors:

$$\frac{\Delta Z}{Z} = \sqrt{\left| \frac{\Delta p}{p} \right|^2 + \left| \frac{\Delta U}{U} \right|^2}. \quad (3.21)$$

The Pythagorean sum used in (3.21) is only strictly correct if the errors in p and U are independent. While this is not true in general, the equation will be approximately correct at impedance maxima and minima, where either $|\Delta p/p| \gg |\Delta U/U|$ or $|\Delta U/U| \gg |\Delta p/p|$. At intermediate values of impedance, where $|\Delta p/p| \approx |\Delta U/U|$, (3.21) will overestimate the error. Since in this work we are mostly interested in impedance maxima and minima, use of (3.21) to estimate the error will lead only to a more conservative redistribution of power.

The variance of the measurement errors may be estimated from experiment or simply assumed equal for each microphone, with possibly an $f^{-0.5}$ -dependence if the noise is of mostly acoustical origin. In this study the sound source was excited with repeated cycles of a periodic signal (see §3.6) and the microphone signals were synchronously acquired in blocks corresponding to one cycle of the excitation signal. Spectra were computed for each block and averaged to reduce noise. The standard deviation in these spectra was used to fit σ_j for each microphone as an exponential function of frequency. The σ_j 's thus obtained were used to calculate the covariance matrix and the error in Z (3.19)–(3.21).

3.6 OPTIMISATION OF THE OUTPUT SIGNAL

To measure an impedance spectrum, an output signal covering all frequencies in the range is required. Some authors use a swept-sinusoid (Dalmont 2001*a*) as the output signal. However, this takes much longer than using a signal with all of the frequency components present, such as white noise or chirps. In the present work, a signal is generated as a sum of components of all sampled frequencies. This signal is applied to the loudspeaker and the impedance and error are calculated.

Initially, the wave is synthesised numerically from components of equal amplitude. To improve the signal-to-noise ratio, the relative phases are adjusted so as to reduce the ratio of the maximum of the sum of sinusoids to the amplitude of each sinusoid, as described by Smith (1995). A wave with a flat power spectrum at the computer does not result in the acoustical wave produced at the reference plane having a flat spectrum, however, because the amplifiers, loudspeaker and connecting conduit all have frequency-dependent responses. These responses could be removed by calculating the power function and multiplying the output spectrum by its inverse (such an approach is used by Wolfe et al. 2001*a*). A flat acoustical spectrum does not, however, produce a flat noise response, because a given head has greater sensitivity to noise at some frequencies than at others. This can be compensated for by multiplying the output spectrum by the correction factor

$$C_1(f) = \frac{\Delta Z}{Z^w} \quad (3.22)$$

(where w is a weighting factor that may be varied to give preference to impedance maxima or minima) and using the resulting waveform in a second impedance measurement. If there are significant nonlinearities in the loudspeaker system, this procedure may be repeated until $C_1(f)$ is tolerably flat, but this is usually unnecessary.

3.7 MATERIALS AND METHODS

3.7.1 The impedance spectrometer

For the experiments described in this paper the configuration of the impedance spectrometer is shown in Figure 3.3. The signal is synthesised on a computer (Macintosh G4) and output via a nominal 24 bit DAC (MOTU 828) to a power amplifier and midrange speaker. A truncated cone helps match the speaker to the measurement head. The impedance spectra are measured between 120 Hz and 4 kHz (a range that encompasses the fundamental frequency of all of the notes of many wind instruments, such as the clarinet and flute, and includes their cutoff frequency). It is easy to extend the frequency range of this technique to other instruments, e.g. the didjeridu (Smith et al. 2007).

A brass measurement head was used with an inner diameter of 15.0 mm and with a 6 mm wall thickness. The brass construction and relatively massive design serve to lessen mechanical conduction of sound and reduce any temperature fluctuations as the head is handled during an experiment.

Three 1/4-inch condenser microphones (Brüel & Kjær 4944 A) are mounted in the impedance head, perpendicular to the cylindrical axis. A 1 mm hole couples each microphone to the

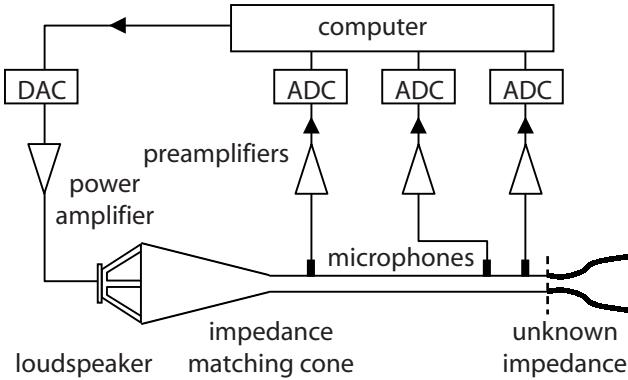


Figure 3.3: The experimental set-up. Calibration used two or three resonant-free calibration loads (see calibration loads (d) in Table 3.2).

waveguide. The compliance of these microphones (equivalent to an air volume of 0.25 mm^3 at 250 Hz) is negligible compared to that of the volume of air between the microphone and coupling hole, which together with the mass of air in the coupling hole forms a Helmholtz resonator with a natural frequency of 6.8 kHz . The coupling hole should be small enough so that the pressure wave is sampled over a distance small compared to its wavelength, and (for measurements using fewer than three calibrations) so that the impedance head is cylindrical, to a good approximation. On the other hand, it should be large enough so that the Helmholtz frequency of the microphone and coupling is much higher than the highest measured frequency. The chosen size of 1 mm represents a compromise between these competing considerations. The three microphones are mounted at $10, 50$ and 250 mm from the reference plane. With the microphones positioned thus, a singularity occurs at $\lambda = 80 \text{ mm}$, which for $c = 345 \text{ m s}^{-1}$ corresponds to a frequency of 4.3 kHz (outside the frequency range of interest).

The signals from each microphone are preamplified and adjusted for calibrated gain by a Brüel & Kjær Nexus conditioning amplifier (2693-0S4) and digitised and recorded by the MOTU interface and the AudioDesk software package. Waveforms are sampled at 44.1 kHz throughout and the output waveform is synthesised at 2^{14} points (giving a frequency resolution of 2.7 Hz). To improve the signal-to-noise ratio the output is cycled repeatedly (100 cycles are typical, resulting in a total measurement time of 37 s) and the recorded signals are averaged. Fourier transforms are performed on the averaged data using the built-in functions in MATLAB.

3.7.2 Calibration loads

The calibration loads (d) in Table 3.2 were used—a quasi-infinite impedance (brass plate), an almost purely resistive impedance (very long pipe), and a quasi-infinite flange.

As pointed out by Dalmont (2001b), in most applications it is sufficient to assume that the admittance of a rigid wall is equal to zero (infinite impedance). Dalmont gives $Z_0 Y_{\text{rigid}} = 9.6 \times 10^{-6} \sqrt{f}(1 + i)$ for the reduced admittance of a rigid wall at 20°C . For a tube of radius 10 mm at 100 Hz , the imaginary part of this admittance corresponds to a length correction of 0.05 mm and the real part to viscothermal dissipation on a length of 3 mm .

The almost purely resistive impedance is a straight PVC pipe of length 97 m and 15 mm internal diameter. The pipe is capped at its far end and filled with a small length (approx. 100 mm)

of acoustically-absorbing wool. If one assumes a fully reflective termination, i.e. neglecting the low reflection termination, at 120 Hz the reflected wave returns with a loss of at least 76 dB (Fletcher & Rossing 1998). The actual loss at 120 Hz will be greater than this lower-bound due to absorption by the acoustic wool. The loss will also increase at higher frequencies due to viscothermal effects. Thus if there were equal power at each of the approx. 1400 frequencies, each reflected component would lie below the effective resolution of the ADC (~ 105 dB).

The quasi-infinite flange is a square perspex plate of side 600 mm in the centre of which is a hole for mounting on the measurement head (for the end effect of a square flange see Dalmont et al. 2001). Over the frequency range of interest the impedance of a flange is lower than that of the resistive impedance load and so these three loads give complementary information. Preliminary experiments have shown that the inclusion of the third load changes the calibration very little, thereby justifying the assumptions made about propagation in the waveguide, and so it was omitted for most measurements.

3.8 RESULTS AND DISCUSSION

3.8.1 Effect of optimising the output signal

As discussed earlier, equal distribution of energy among frequencies does not result in equal distribution of errors in the measurements, and the size of the largest error can be substantially reduced by adjusting the output spectrum for particular circumstances and loads.

Figure 3.4 shows the magnitude of and (absolute) fractional error in the measured impedance for three sequential impedance measurements. The error was calculated using (3.19)–(3.21). In Figure 3.4 (a), the impedance Z of the resistive impedance load is measured with a non-ideal source. The measured impedance spectrum is not completely flat since this measurement was performed before calibration. The error function $\Delta Z/Z$ in Figure 3.4 (a) has broad features corresponding primarily to loudspeaker and conduit resonances. The output spectrum is multiplied by the error function in Figure 3.4 (a) and used to measure an open pipe (Figure 3.4 (b)). The output spectrum is again multiplied by the error function and the pipe is measured a second time. The error in the resulting measurement (Figure 3.4 (c)) is uniformly distributed over frequency. The errors shown in Figure 3.4 (b) and (c) have been divided by a factor of 10 (the signal amplitude was deliberately kept smaller than optimum to increase the apparent errors).

It is worth noting that in the uncorrected measurement (b) very high or very low impedances are measured with a greater error than are impedances close to Z_0 . This is unfortunate in the case of musical instruments where we are particularly interested in impedance maxima and/or minima. On subsequent iterations, more power is put into these frequencies and the error is thereby reduced. As can be seen in Figure 3.4 (c), the error increases for impedances close to Z_0 (the error is also more apparent here since the slope of the curve is not as steep as at impedance extrema). However, the maximum error over the entire spectrum is reduced by a factor of approx. 10 through this optimisation procedure.

Also shown in Figure 3.4 is the sound pressure spectrum measured by the microphone closest to the reference plane for each measurement. This is relatively uniform over frequency for

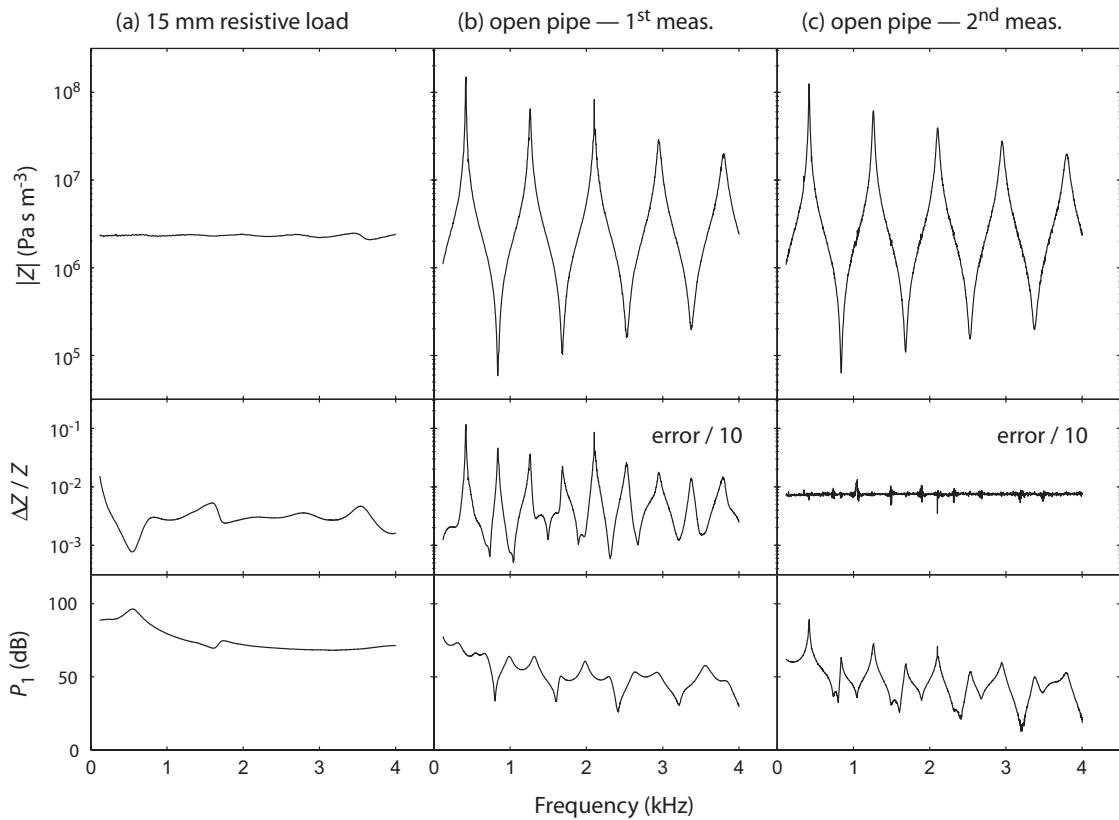


Figure 3.4: Optimising the output signal for measurements on 15 mm pipes. Impedance magnitude (*upper panel*) and fractional error (*middle panel*) are shown for (a) the 15 mm resistive impedance load measured with a non-ideal source and for an open pipe of length 200 mm measured with a deliberately very low signal (b) before and (c) after optimisation. The error spectra in (b) and (c) have been scaled for comparison with (a). Also shown for each measurement is the sound pressure in dB *re* 20 μ Pa (*lower panel*) as measured by the microphone nearest to the reference plane.

the measurement of the resistive impedance and has minima corresponding to nodes of the standing wave for measurements of the 200 mm closed pipe.

The choice of the exponent w in $C_1(f)$ (3.22) determines which part of the spectrum will be measured with the greatest precision. For $w = 1$, the fractional error will be constant for all impedances. For $w > 1$, the impedance minima will be determined with greater precision, and for $w < 1$ the impedance maxima will be given preference. The case where $w > 1$ is particularly useful for measuring the impedance of instruments in the flute family, which play near impedance minima, whereas $w < 1$ is useful for reed and lip valve instruments such as the clarinet which play near impedance maxima.

In some cases (such as when measuring calibration loads) it is not desirable to use the function $C_1(f)$ to modify the output spectrum, but we may still like to compensate for the system responses and the ‘singularity factor’ of the head. In these cases, we modify the output spectrum to ensure that the acoustic energy density at the reference plane is the same as it would be during a (hypothetical) measurement of Z_0 with $\Delta Z/Z = K$ where K is independent of frequency. The acoustic energy density at the reference plane during a measurement of an impedance Z is proportional to $\epsilon = \frac{1}{2}(|p|^2 + |Z_0 U|^2)$. For a measurement of Z_0 ,

$$\epsilon_0 = |Z_0 U|^2 = \frac{|\Delta p|^2 + |Z_0 \Delta U|^2}{K^2}, \quad (3.23)$$

where (3.21) was used for K with the substitution $p = Z_0 U$. The correction factor $C_2(f)$ used to modify the output spectrum will be proportional to the square root of the energy ratio $\frac{\epsilon_0}{\epsilon}$. We use

$$C_2(f) = K \sqrt{\frac{\epsilon_0}{\epsilon}} = \sqrt{2} \sqrt{\frac{|\Delta p|^2 + |Z_0 \Delta U|^2}{|p|^2 + |Z_0 U|^2}}, \quad (3.24)$$

where the factor K has no effect on the output waveform (being independent of frequency) but is used to ensure that $C_2(f) = \Delta Z/Z$ when $Z = Z_0$.

3.8.2 Effect of calibration

Figure 3.5 illustrates the effects of calibrating with varying numbers of known calibration loads. Measured impedance spectra are shown for a closed 15 mm pipe, 200 mm long. To simulate the effect of using unmatched microphones, in Figure 3.5 (a) and (b) the signals from the second and third microphones were scaled by 1.2 and 0.8, respectively. The measured impedance spectrum before calibration (i.e. assuming unity gain for each microphone) is shown in Figure 3.5 (a), while in Figure 3.5 (b) the impedance was calculated after calibrating on the quasi-infinite impedance load (using (3.7) and (3.10)). The pre-calibration measurement deviates significantly from the theoretical impedance, while calibrating with only one load (b) improves the accuracy significantly. In Figure 3.5 (c) the measured impedance spectrum using the matched microphone signals is shown before and after calibration with two known loads. While the two spectra are similar in many respects, the size and frequency of extrema (particularly maxima) are significantly different. For this measurement, adding the flange calibration made very little difference. The measurement made without using the two calibration loads shows similar features to those of the precise measurement over most of the range, but has noticeable errors in the magnitude of extrema (particularly maxima) and smaller errors in the frequencies at which

they occur. In Figure 3.5(d) the measurement on the 200 mm closed pipe using two calibration loads is compared with the theoretical impedance and the fractional difference between these two measurements is shown in (e). The peaks in this difference function at impedance extrema are likely due to slightly greater attenuation in the measured impedance than is predicted by the theory for viscothermal losses at walls that are ideally smooth.

3.8.3 Conclusions and practical considerations for measurement

The combination of three microphones, three non-resonant calibrations and spectral shaping can give precise measurements over a wide range of frequencies and impedance. It has the added advantage that no assumptions need be made about the microphone characteristics or about the exact geometry of the impedance head. Furthermore, there is no need to invoke a theoretical model for waveguide losses.

In many practical measurement situations, however, the reduced performance of a simpler combination might still be appropriate.

Thus two microphones, rather than three, would often be sufficient for a smaller frequency range (perhaps 2–3 octaves, depending upon the precision required).

In many practical cases, fewer than three non-resonant calibrations might be sufficient, albeit with some loss in performance. Thus if a well-defined cylindrical impedance head is used, and if the perturbations of the cylinder by the microphones are sufficiently small, it is possible to remove one calibration (this is a consequence of the good theoretical model available for a cylindrical waveguide). It is also possible to remove one calibration if the characteristics of the microphones are already known with a high degree of precision.

The redistribution of power in the source function (spectral shaping) can improve the signal to noise ratio at extrema by a factor of 10 or more. Whether or not this feature is needed depends on the size of errors that may be tolerated in the measurements. Further, this feature would be less important if the unknown impedance has no strong resonances.

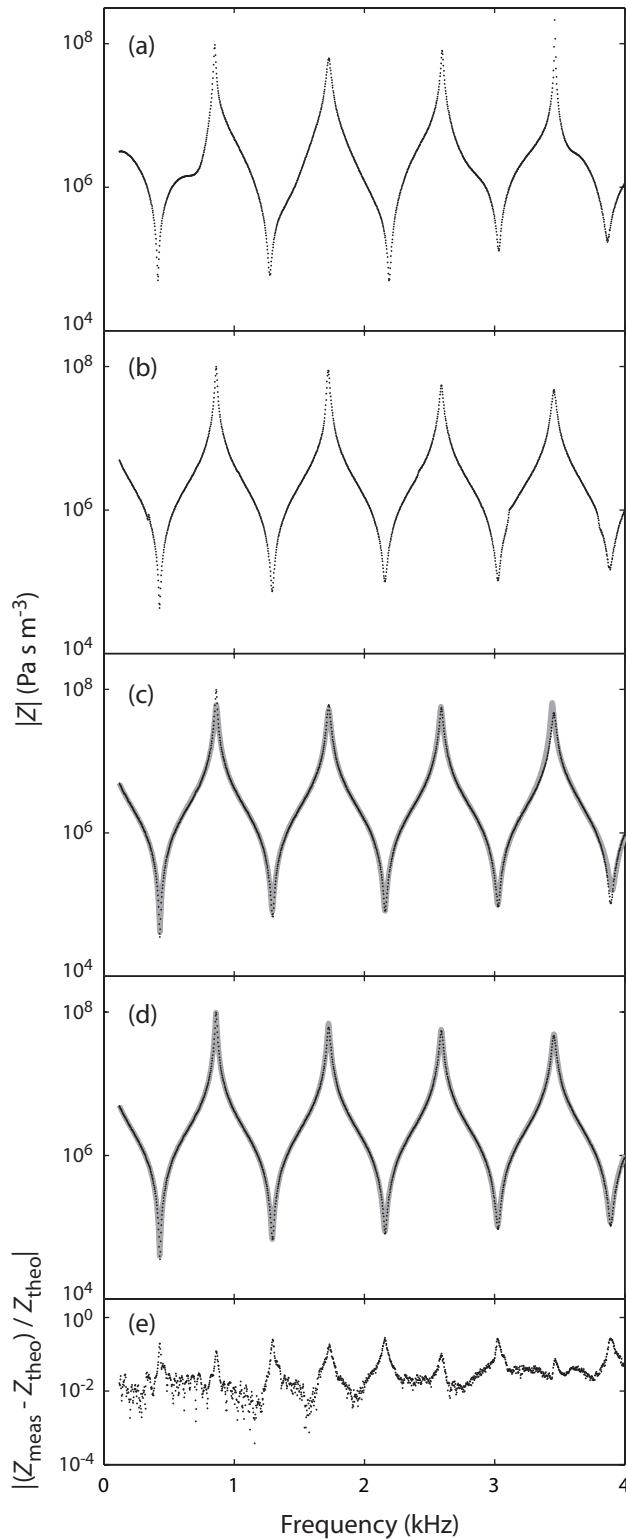


Figure 3.5: The input impedance for a closed 15 mm pipe, 200 mm long. In (a) and (b) the signals from microphones 2 and 3 were multiplied by 1.2 and 0.8 respectively to deliberately introduce errors. The impedance was measured before calibration (a) and after calibrating using one known load (b). In (c) the impedance is given for matched microphones before (*grey, solid line*) and after (*black, dotted line*) calibration with two known loads, and in (d) the calibrated measurement from (c) (*black, dotted line*) is compared with the theoretical impedance for an ideal tube (*grey, solid line*). The fractional difference between this theory and the measurement is shown in (e).

Chapter IV

Finger hole impedance spectra and length corrections

4.1 INTRODUCTION

Several theoretical and experimental investigations have examined the impedance effects and equivalent circuits of open and closed tone holes. From these studies length corrections have been derived that are in most cases sufficient for predicting the effect of a tone hole on the tuning of a woodwind instrument—formulae for many of these length corrections are given in Chapter 2. However, most of these studies are limited to simple keyed tone holes, formed with a chimney that meets the key at a plane. No simple formulae exist for the equivalent circuit elements to use in the case of finger holes such as are found on classical flutes and recorders. These finger holes are drilled directly through the wall of the instrument, are often undercut and are closed with the player's fingers. In some cases such holes are as large as the tip of a finger, and when closed the curved pad of the finger protrudes a significant distance into the bore of the instrument. In this chapter the equivalent circuit for a finger hole is measured over a wide range of geometrical parameters. The associated length corrections are calculated and compared with fit-formulae in the literature. An extensive literature search found no length corrections to use for a closed finger hole—for this case fit-formulae are derived from the measurements in this chapter.

In the simplest one-dimensional model, an open tone hole on a woodwind instrument effectively cuts the instrument off at the position of the hole—no acoustic waves penetrate further along the instrument and in the electro-acoustical model the hole acts as a short-circuit to ground. In this model closed holes have no effect. However, only for the case of large holes at low frequencies is this approximation in any way valid. A slightly more complex model takes into account the non-zero inertance of the air in an open tone hole. This inertance is greater for longer holes but is still significant for short holes due to the radiation impedance of a flanged hole. Two acoustic paths are then possible: through the tone hole to the radiation field ('ground') or past the tone hole and into the rest of the flute. Thus the tone hole may be represented in the transmission line circuit as a shunt impedance to ground. Closed holes are represented likewise; however in this case the shunt impedance is a compliance, representing the extra air volume beneath the closed key. Further refinements may be made to the equivalent circuit of a tone hole. One important refinement is to add a series impedance to the circuit. This impedance (mostly a negative inertance) represents the effect of flow widening at the tone hole.

The effect of an open or closed tone hole on the resonant properties of woodwind instruments may be described according to its effect in two idealised situations: when the hole is at a pressure antinode and there is maximal acoustic flow to the outside air through the hole; and when the hole is at a flow antinode and there is no net acoustic flow through the hole. In the

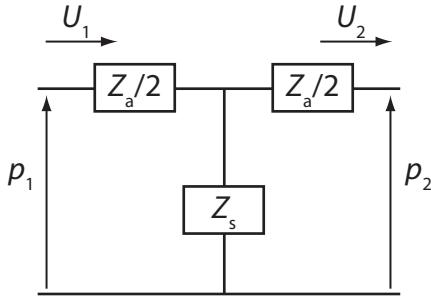


Figure 4.1: A T-junction representation of a finger hole showing pressure p and volume flow U at the input (subscript 1) and output sides of the hole (subscript 2).

electro-acoustic analogy the hole may be represented by the T-junction shown in Figure 4.1, comprised of the two impedances Z_a (the ‘series’ impedance) and Z_s (the ‘shunt’ impedance). This lumped-element model will accurately describe the acoustic effect of the hole provided the wavelength is significantly greater than the hole dimensions, and provided the hole is symmetrical with respect to the bore axis. Asymmetrical holes may be represented by a T-junction with three different impedances (Poulton 2005). Although all of the holes measured in this chapter are symmetrical, the methods may be easily extended to measure the shunt and series impedances of asymmetrical tone holes.

Coltman (1979) measured the shunt reactance for flute tone holes using a tube closed at one end by a piston driver and at the other by a rigid microphone. The tone hole and key mechanism were placed at the centre of the tube. In the two lowest modes of this system (the so-called ‘slosh’ and ‘butt’ modes) the two quarter-wave ends oscillate in either the same or opposite directions. In the ‘slosh’ mode the resonance frequency is almost unaffected by the presence of the hole (since for this mode there is a pressure node at the hole position), whereas in the ‘butt’ mode some acoustic flow escapes through the hole and the frequency is reduced. The reactance of the tone hole is derived easily from these two resonance frequencies. With such a simple system Coltman found the open hole shunt reactance (expressed as a length correction) for a wide range of hole–key combinations. Coltman also measured the change in air column effective length caused by closed tone holes. (This is related to the closed hole shunt and series reactances.) Clearly, any frequency-dependence of the end correction cannot be measured with this system.

Keefe (1982a) measured the shunt reactance and resistance and the series reactance of open and closed tone holes. Using a system similar to that of Coltman, Keefe measured both ‘saxophone-’ and ‘clarinet-like’ holes. ‘Saxophone-like’ holes have a diameter much greater than their height while the height of ‘clarinet-like’ holes is equal to or greater than their diameter. The input impedance of a closed pipe with the tone hole in the centre was measured. The tone hole impedance components were derived from the frequency and bandwidth of peaks in the impedance spectrum. Keefe found that the measured reactances agreed with theory (Keefe 1982b) and with the flute data obtained by Coltman (1979). The shunt resistance of an open hole was also measured under linear conditions although, as Keefe points out, the acoustic level is much lower than under playing conditions, when nonlinear losses become important.

In a series of measurements on the open tone hole, Dalmont et al. (2002) used a similar setup to that of Keefe, but included a microphone in the passive tube termination, thereby permitting measurement of the impedance matrix or transfer matrix. Dalmont et al. measured the series and shunt reactances, showing that they are well-described by two theoretical studies (Nederveen et al. 1998, Dubos et al. 1999). Dalmont et al. also measured the shunt resistance, concluding that it may be described by a theory including viscothermal losses in the tone hole and the radiation impedance. Since the transfer matrix was measured, results were obtained for all frequencies in the measurement range, not just those corresponding to an impedance extrema, although for frequencies such that $\sin(kL) \approx 1$, where L is the length of the measurement tube, flow through the tone hole is essentially zero and the shunt impedance is not ‘seen’ by the apparatus.

Dalmont et al. (2002) also measured the open hole equivalent circuit at large amplitude using a two-microphone impedance head with an extra microphone downstream of the tone hole. This system was capable of delivering a much greater acoustic flow than the volume flow source impedance head, allowing determination of nonlinear losses in the vicinity of the tone hole. At high acoustic flow, resistances proportional to the velocity must be added to the series and shunt impedances. The two-microphone method was not used for low amplitude measurements since the authors were unable to obtain an accuracy on the length corrections lower than 10% of the radius of the main tube.

4.2 MATERIALS AND METHODS

In the following experiments, the finger hole is placed between a pair of two-microphone impedance heads so that the pressure and flow are measured at both sides of the hole (Figure 4.2). From these measurements the shunt and series impedances are determined. The impedance heads are calibrated individually using the methods of Chapter 3 and the signals applied to each loudspeaker are adjusted to optimise the measurement of either Z_s or Z_a .

4.2.1 Modified impedance spectrometer

Two impedance heads with bore diameter of 15 mm are used, each equipped with two microphones; one 10 mm from the reference plane and the other 50 mm. However, to accommodate the finger hole, the reference plane for each head is shifted outwards from the microphones by 50 mm and the finger hole is formed in the centre of a tube section of length 100 mm.

Two nominally identical midrange loudspeakers are used as the excitation sources. Care is taken to ensure that the loudspeakers are wired with the same polarity. It is preferable if the distance from the reference plane to the loudspeaker is similar for each head, but this is not essential. The voltage signal applied to one speaker is V_r and to the other $\frac{V}{r}$. The complex factor r determines the splitting of the signal between each loudspeaker. Different loudspeaker efficiencies may be accounted for by calibration (see §4.2.3.2).

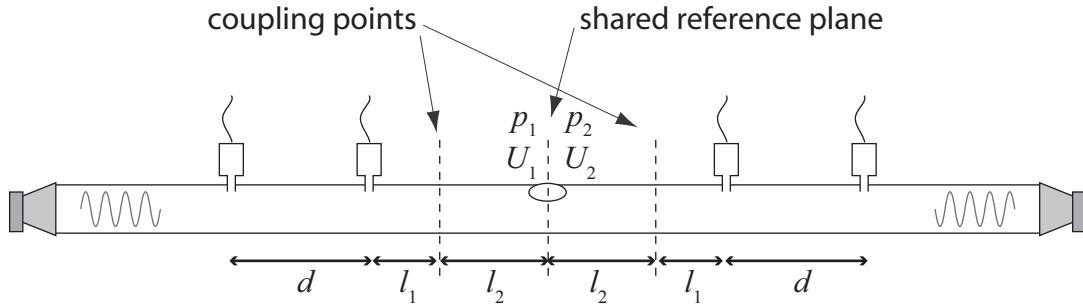


Figure 4.2: The experimental set-up for measurement of finger hole properties using two impedance heads (not to scale). In the diagram $d = 40$, $l_1 = 10$ and $l_2 = 50$ mm. The pressure p_1 and volume flow U_1 are defined at an infinitesimal distance to the left of the finger hole (treated as a lumped element) and p_2 and U_2 are the pressure and volume flow at an infinitesimal distance to the right of the finger hole.

4.2.2 Calculation of Z_s and Z_a

For an optimised measurement of the shunt impedance Z_s , we require maximal flow through the finger hole to the external field. In other words, we require a pressure antinode at the x -position occupied by the finger hole (equivalent to Coltman's 'butt' mode). This condition is approximately satisfied by setting $r = 1$ (the loudspeakers oscillate in-phase). For an initial measurement, the frequency spectrum of V may be the standard excitation function with frequency-independent modulus and randomised phase (as described in §3.6), or may be derived from calibration measurements on each measurement head. Using these initial values, the microphone responses are recorded and the pressure and flow on each side of the finger hole are determined. Conversely, for a measurement of the series impedance Z_a , we require a flow antinode at the hole position. This is approximately achieved by setting $r = i$ (the loudspeakers oscillate with a phase difference of π).

As shown in Figure 4.2, each impedance head measures the pressure and flow at a third reference plane, on the assumption of plane-wave propagation through a cylindrical pipe. Thus the tone hole at the reference plane is treated as a lumped element, having no spatial extent but (possibly) causing discontinuities in pressure and flow. Given pressure and flow at the finger hole as measured from one side p_1 and U_1 , and pressure and flow as measured from the other side p_2 and U_2 , the shunt and series impedance may be derived by applying Kirchhoff's laws to the circuit shown in Figure 4.1. The shunt impedance is then given by

$$Z_s = \frac{p_2 U_1 - p_1 U_2}{U_1^2 - U_2^2} \quad (4.1)$$

and the series impedance by

$$Z_a = 2 \frac{p_1 - p_2}{U_1 - U_2}. \quad (4.2)$$

Note that for $p_1 = -p_2$ and $U_1 = -U_2$, Z_s is indeterminate and for $p_1 = p_2$ and $U_1 = U_2$, Z_a is indeterminate.

4.2.3 Calibration

4.2.3.1 Calibration of impedance heads

Each impedance head is calibrated separately against known impedance loads, using the methods described in §3.4.2 with some modifications. Most importantly, since the reference plane for each head is 50 mm further from the microphones than the coupling point, the ‘infinite impedance’ and the ‘infinite flange’ calibration loads are in reality pipes of length 50 mm with the appropriate termination. Further, in §3.4.2, we were only interested in measuring the input impedance rather than pressure and flow. Consequently, only three calibration loads were required and one element (A_{11}) in the matrix \mathbf{A} was left undetermined, with no effect on impedance measurements. In determining the equivalent circuit for a finger hole, pressure and flow differences enter into the equations, so the impedance heads should be calibrated absolutely for pressure and flow.

This can be achieved by flush-mounting a reference microphone in the infinite impedance calibration load. Assuming that the microphone has negligible compliance, the elements A_{j1} are derived from the pressure p measured at the face of this impedance load as

$$A_{j1} = b_j / p \quad (4.3)$$

where, as in (3.7), b_j is the signal from microphone j ($j = 1, 2, \dots, n$ where n is the number of microphones). The remaining elements in \mathbf{A} are determined using the methods of §3.4.2 from measurements of the semi-infinite pipe and the infinite flange calibration loads.

4.2.3.2 Calibration of entire system

The two heads are attached to either end of a 100 mm tube with no finger hole (thereby sharing the same reference plane). For this configuration, $p_1 = p_2$ and $U_1 = -U_2$ (the sign difference is a consequence of the definition of positive flow being *into* the measurement load). However, allowing for small differences in the microphones comprising each array, the measured pressures and flows may be somewhat different. This is accounted for by making small changes to the matrix \mathbf{A} for each head.

A measurement of each of Z_s and Z_a is performed on the ‘no-hole’ system. In each case, the pressure at the reference plane p is taken as the mean of the pressure seen by each measurement head at the reference plane. Likewise the flow U is taken as the mean of the flows (with a sign change due to the way flow is defined). Then for each head we have the following matrix equation:

$$\mathbf{A} \begin{bmatrix} p_s & p_a \\ Z_0 U_s & Z_0 U_a \end{bmatrix} = \begin{bmatrix} \mathbf{c}_s & \mathbf{c}_a \end{bmatrix}, \quad (4.4)$$

where \mathbf{c}_s and \mathbf{c}_a are the vectors of microphone signals obtained during the measurement of Z_s and Z_a respectively. This equation may be solved for the matrices \mathbf{A} for each head.

The splitting ratio r may also be improved by calibration. As mentioned above, the voltage signal applied to each loudspeaker is Vr or $\frac{V}{r}$. The pressure and flow at the reference plane, p

and U , are then each some linear combination of the two loudspeaker signals:

$$p = \alpha Vr + \frac{\beta V}{r} \quad (4.5)$$

$$U = \delta Vr + \frac{\gamma V}{r}. \quad (4.6)$$

(The proportionality constants may be calculated using models of each source, impedance head and any connecting conduit, but this is unnecessary for the present.) For a measurement of Z_s on the ‘no-hole’ system, we require $U = 0$, or (using (4.6)) $r_s^2 = -\frac{\gamma}{\delta}$. Likewise, for a measurement of Z_a , we require $p = 0$, or (from (4.5)) $r_a^2 = -\frac{\beta}{\alpha}$. The four proportionality constants may be determined from the four measurements p_s and U_s (measured with $r = r_s$, initially 1) and p_a and U_a (measured with $r = r_a$, initially i). The adjusted splitting ratios r_s' and r_a' are then given by

$$r_s' = \sqrt{r_s r_a \frac{r_s V_s U_a - r_a V_a U_s}{r_a V_s U_a - r_s V_a U_s}}, \quad (4.7)$$

and

$$r_a' = \sqrt{r_s r_a \frac{r_s V_s p_a - r_a V_a p_s}{r_a V_s p_a - r_s V_a p_s}}. \quad (4.8)$$

Note that the loudspeakers are assumed to be linear in the above calculation of the splitting ratios. To reduce the effect of loudspeaker non-linearity very strong resonances in the system should be avoided so that the magnitudes of V_s and V_a are similar. For this reason, acoustic wool is placed between each microphone array and loudspeaker.

4.2.4 Measurement optimisation

After an initial measurement of Z_s or Z_a , we can change the output spectrum V to optimise the measurement in a manner similar to that described in §3.6.

Some simplifying assumptions make this easier. For a measurement of Z_s , we assume that $Z_a = 0$. Then $Z_s = \frac{p}{U}$ where $p = \frac{p_1 + p_2}{2}$ and $U = U_1 + U_2$. For a measurement of Z_a , we assume that $Z_s = \infty$ (when there is a pressure node at the hole there is zero flow through the hole and Z_s is not ‘seen’). Then $Z_a = \frac{p}{U}$ where $p = p_1 - p_2$ and $U = \frac{U_1 - U_2}{2}$. The errors in p and U , Δp and ΔU , are obtained in both cases by straight-forward propagation of errors, as is the error ΔZ in Z (for simplicity the subscript has been dropped here).

Having determined the error in Z , the output spectrum is modified by multiplying with either of the correction functions C_1 (3.22) or C_2 (3.24).

4.2.5 Frequency range and dynamic range

Impedance spectra were measured between 1 and 3 kHz. This frequency range was chosen as it encompasses the upper range of woodwind instruments, where the finger hole series and shunt impedances have most effect. Below 1 kHz the series and shunt impedances are near or beyond the limits of the dynamic range of the system, at least for small holes. Also, the microphone separation of the impedance heads measures impedance optimally at the centre of this range. During measurements on finger holes, the control pipe with no hole was measured repeatedly to check for drift and to gauge the dynamic range of the system. The system dynamic

range depends on (among other things) random noise, the accuracy of machining of the experimental tubes and drift in temperature and humidity. The shunt impedance of the control tube measured consistently above $2 \times 10^8 \text{ Pa s m}^{-3}$ (theoretically infinite) and the series impedance measured consistently below $2 \times 10^4 \text{ Pa s m}^{-3}$ (theoretically zero). These figures represent the limits of the dynamic range (80 dB) of the system.

4.2.6 Investigated hole geometries

Photographs of the finger holes measured are shown in Figure 4.3. The finger holes were drilled in tubes of PVC conduit of length 100 mm and inner diameter 15 mm. The hole diameters ranged from 1.5 mm to 15 mm in 1.5 mm steps. A set of holes was formed in the unaltered conduit, which has a wall thickness of 2.5 mm. Another set of tubes was turned on a lathe to reduce the wall thickness in the vicinity of the hole to 1.0 mm and the wall thickness of another set of tubes was increased to 5.0 mm by inserting each tube into a larger tube chosen to ensure a tight fit. Thus a range of hole geometries was measured, covering most geometries likely to be encountered on a classical flute and many other woodwinds.

4.2.7 Measurement conditions

Z_s and Z_a spectra were measured for both open and closed finger holes. I used my own fingers to close the holes, using light to moderate finger pressure. For some finger holes, one or more of the four impedance spectra were omitted from the measurements, since the dynamic range of the system was not sufficient.

4.3 RESULTS AND DISCUSSION

4.3.1 Open finger holes

4.3.1.1 Shunt impedance and radiation length correction

The shunt impedance of an open hole can be represented as the sum of the inner radiation impedance and the input impedance of a cylindrical pipe section terminated by the radiation impedance of a hole with cylindrical flange. (The length of the cylindrical pipe section is given by the wall thickness plus the matching volume length correction given in (2.34).)

The shunt impedance for open holes of length 2.5 mm and differing diameters is shown in Figure 4.4. Similar spectra were obtained for holes of length 1.0 and 5.0 mm (not shown here).

Using (2.39) for the inner radiation length correction, (2.34) for the the matching volume length correction and (2.12) for the transfer function of the hole, we can derive the radiation impedance for a hole flanged by a cylinder and compare the associated length correction to the literature. This is done for holes of length 1 mm in Figure 4.5 and for holes of lengths 2.5 and 5.0 mm in Figures 4.6 and 4.7.

The scatter in the experimental results shown in Figures 4.5 to 4.7 is greatest for small holes, where the measured impedance is much greater than the characteristic impedance of the measurement heads, and the effects of systematic errors are most prominent. However, the fit formula used (2.29) is consistent with the experimental results at least for $0.5 \leq b/a \leq 1.0$. The formula used by Dalmont et al. converges to the well-known length correction for an infinitely

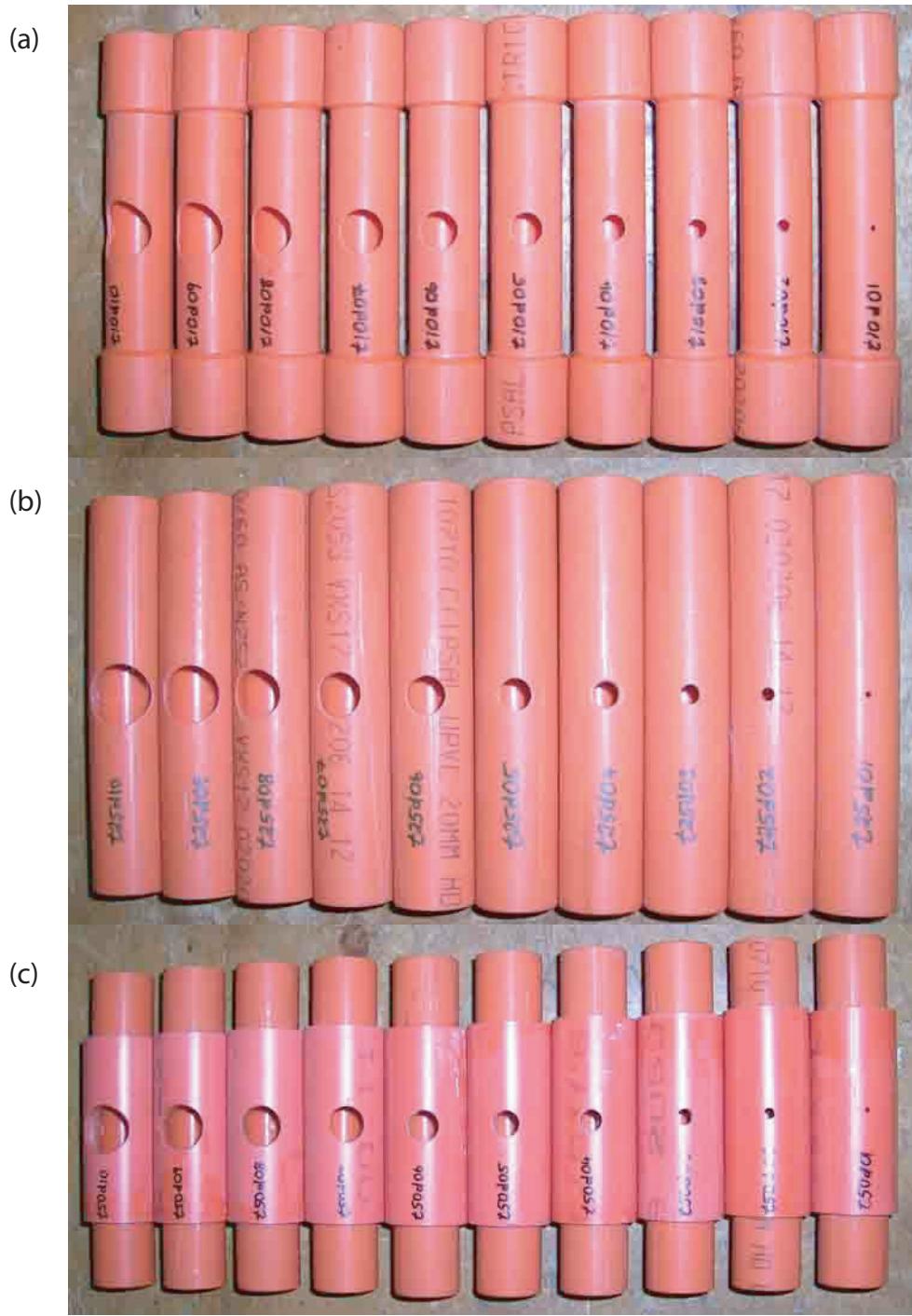


Figure 4.3: Photographs of the finger holes measured. The hole length t varied from (a) 1.0 mm, to (b) 2.5 mm and (c) 5.0 mm while the radius ratio b/a varied from 0.1 to 1.0. The tube inner radius a was 7.5 mm.

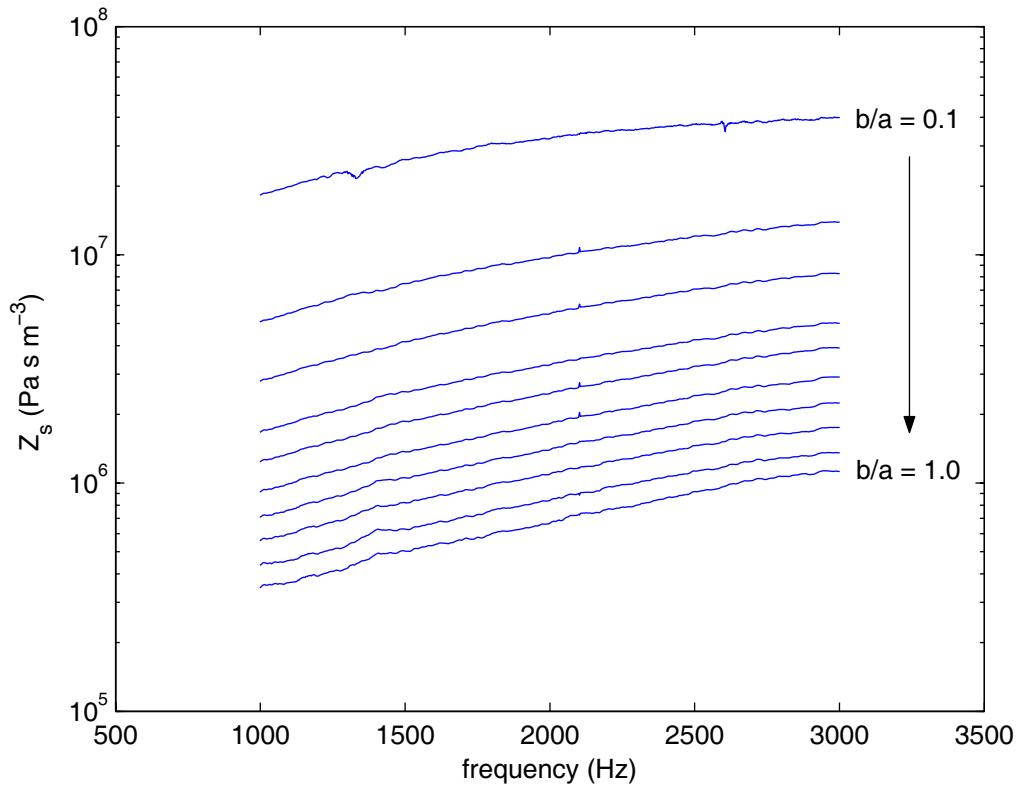


Figure 4.4: The shunt impedance Z_s for open finger holes of length 2.5 mm and $b/a = 0.1$ to 1.0 in steps of 0.1, $a = 7.5$ mm.

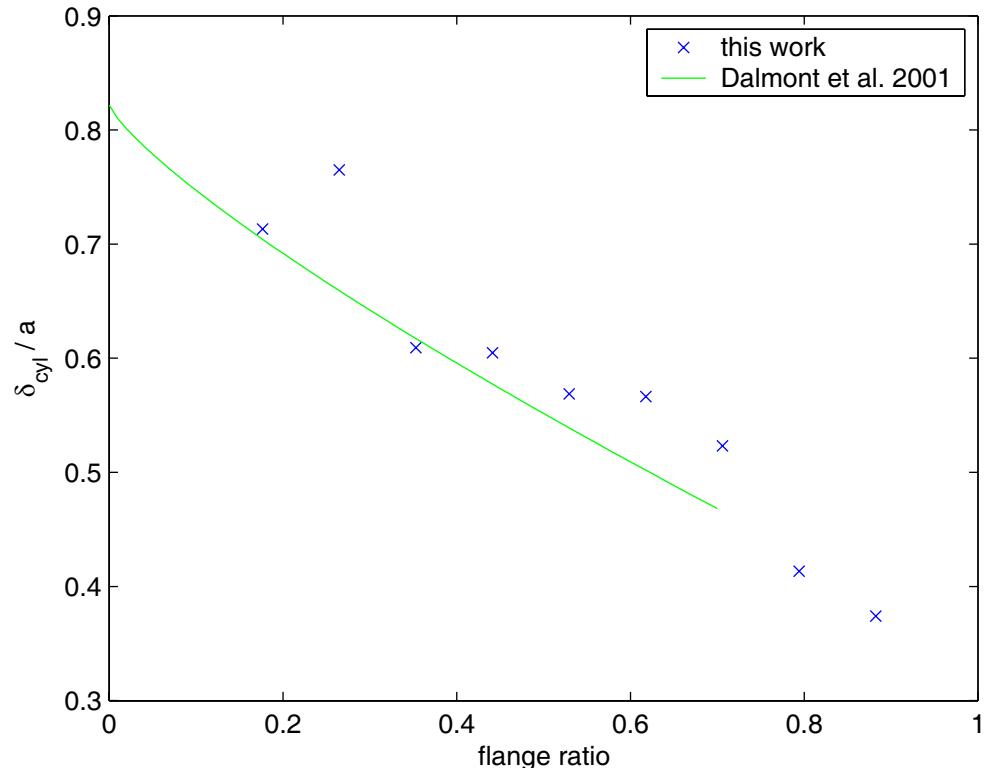


Figure 4.5: The measured length correction δ_{cyl} for radiation from open finger holes of length 1.0 mm compared to the fit-formula of Dalmont et al. (2001). The 'flange ratio' is the ratio of the radius of the finger hole to the radius of the cylindrical flange.

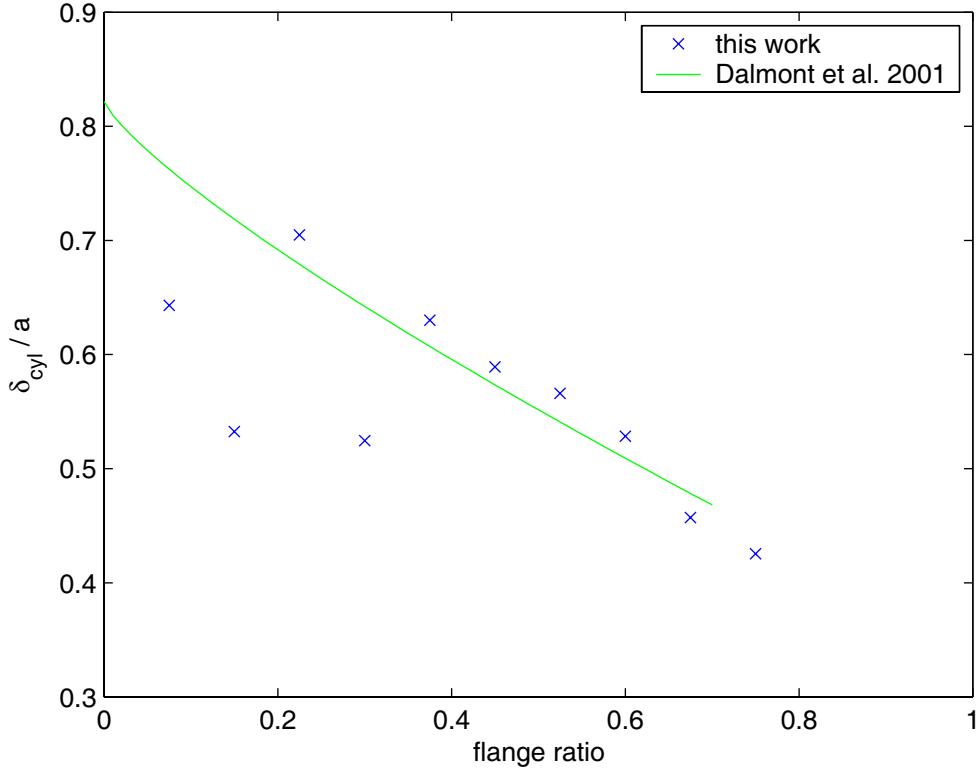


Figure 4.6: The measured length correction δ_{cyl} for radiation from open finger holes of length 2.5 mm compared to the fit-formula of Dalmont et al. (2001). The ‘flange ratio’ is the ratio of the radius of the finger hole to the radius of the cylindrical flange.

flanged pipe as the flange ratio approaches zero. Therefore, the fit formula will be used henceforth.

4.3.1.2 Series impedance and length correction

The measured series impedance for the 2.5 mm holes is shown in Figure 4.8. Only holes with $b/a > 0.5$ are shown, since the series impedance for holes of smaller diameter is below the dynamic range of the measurement system. In the subsequent analysis frequency components below 1.8 kHz were ignored, due to the influence of systematic errors. Likewise, the singularity at approx. 2.1 kHz was removed before analysis.

The series length correction t_a was calculated from the series impedance Z_a using the equation

$$t_a = \frac{Z_a}{ikZ_0}. \quad (4.9)$$

The imaginary part of t_a was discarded and the results were averaged over frequency. The measured length corrections for all sets of finger holes are shown in Figure 4.9. Shown for comparison is the equation of Dubos et al. (1999) for long finger holes (2.37). This equation overestimates the size of the correction for small holes and underestimates it for large holes—however the investigated holes do not satisfy the condition $t > b$ so the equation is unlikely to fit exactly. Dubos et al. (1999) also gives an equation which scales with t for open tone holes (2.35), but this equation was found to overestimate the size of t_a , particularly at small t . Dalmont et al. (2002) question the use of this formula, since for short holes the internal and external discontinuities

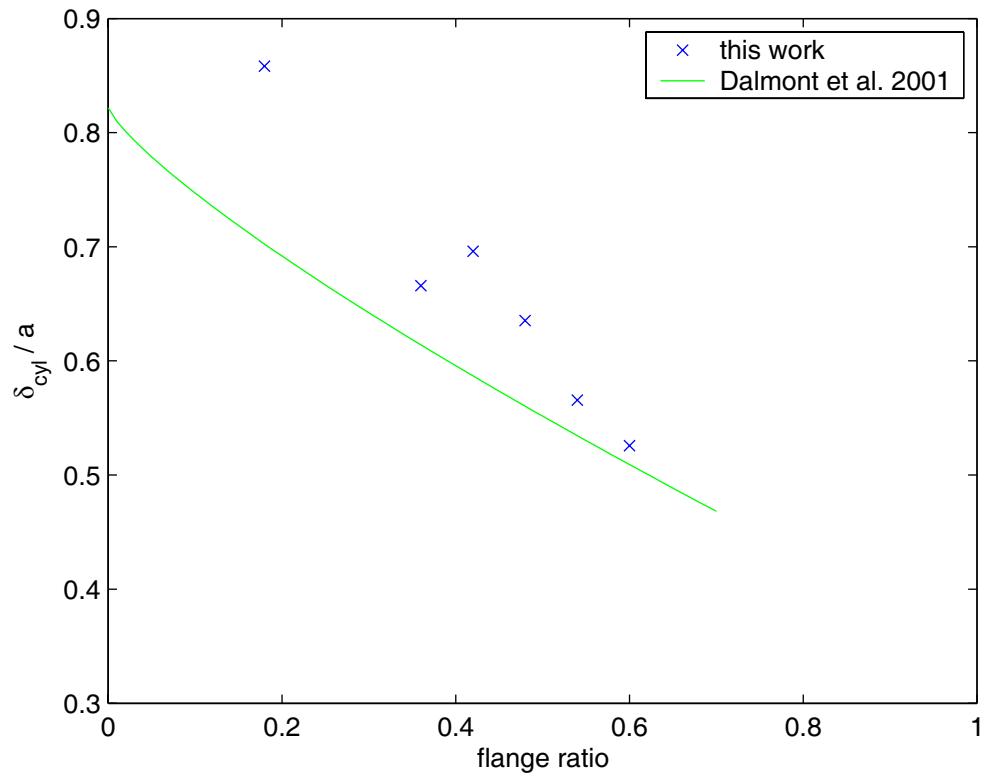


Figure 4.7: The measured length correction δ_{cyl} for radiation from open finger holes of length 5.0 mm compared to the fit-formula of Dalmont et al. (2001). The ‘flange ratio’ is the ratio of the radius of the finger hole to the radius of the cylindrical flange.

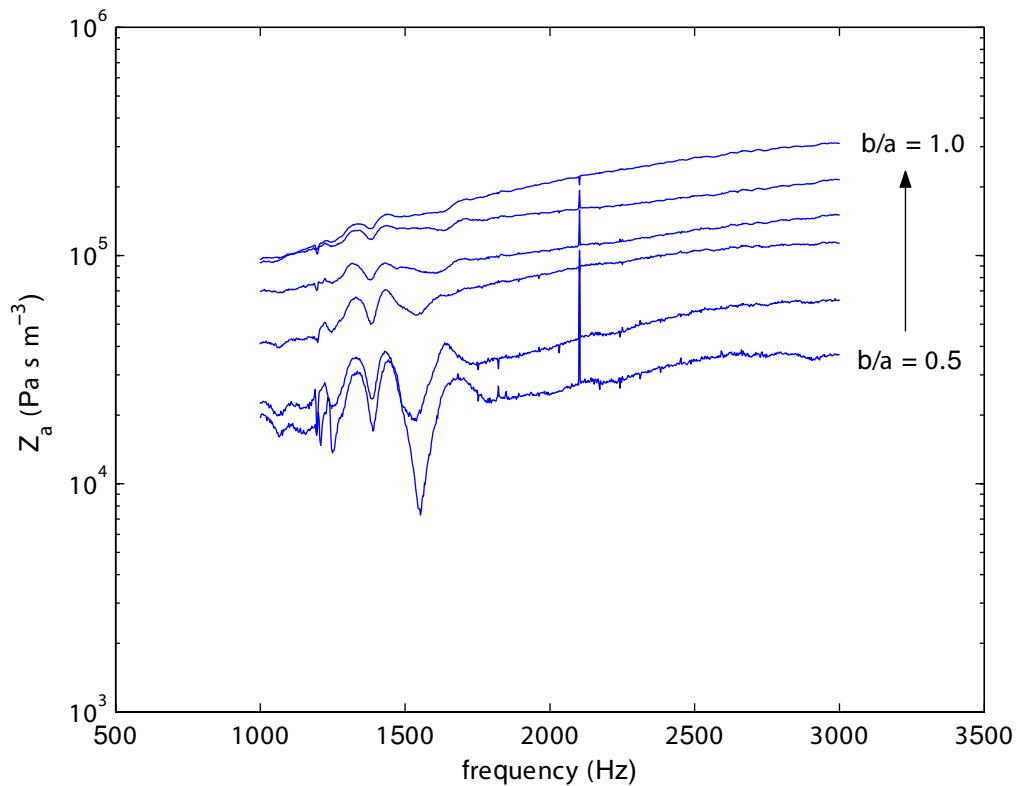


Figure 4.8: The series impedance Z_a for open holes of length 2.5 mm.

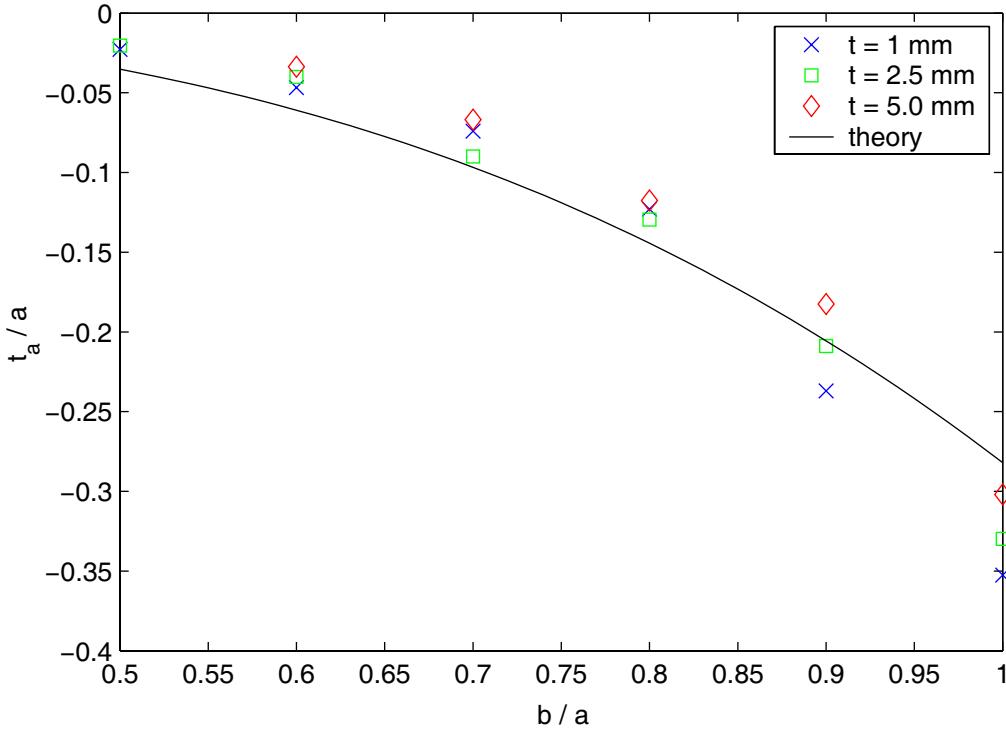


Figure 4.9: The series length correction t_a for open holes compared with (2.37).

are coupled.

Figure 4.9 indicates that some modifications to the fit formula (2.37) may be required for open finger holes but (2.37) is probably sufficient to use in the current work.

4.3.2 Closed Finger holes

4.3.2.1 Shunt impedance and finger length correction

The shunt impedance of a closed finger hole may similarly be considered as the sum of the inner radiation impedance and the input impedance of the hole tube section terminated by the load Z_{finger} , the impedance of the player's finger. The load impedance is derived from the measurement of Z_s as in §4.3.1.1 and the finger length correction is given by

$$t_{\text{finger}} = \frac{\cot^{-1}(Z_{\text{finger}})}{-ikZ_0}. \quad (4.10)$$

The correction t_{finger} is expected to be a negative quantity, since the finger protrudes into the hole, increasing in size with hole diameter.

The shunt impedance Z_s for closed finger holes of length 1.0 mm with $b/a = 0.5$ to 1.0 is shown in Figure 4.10. Similar spectra were obtained for hole lengths 2.5 and 5.0 mm. From these spectra, the length correction due to the player's fingers may be derived. Due to the errors below 1.5 kHz, frequencies below this limit were not included in the calculation of the length correction.

The measured finger length correction is shown in Figure 4.11. Some data are missing, due to the measured impedance being above the dynamic range of the measuring system. (For example, if the volume occupied by the finger is equal to the volume of the hole, then the closed

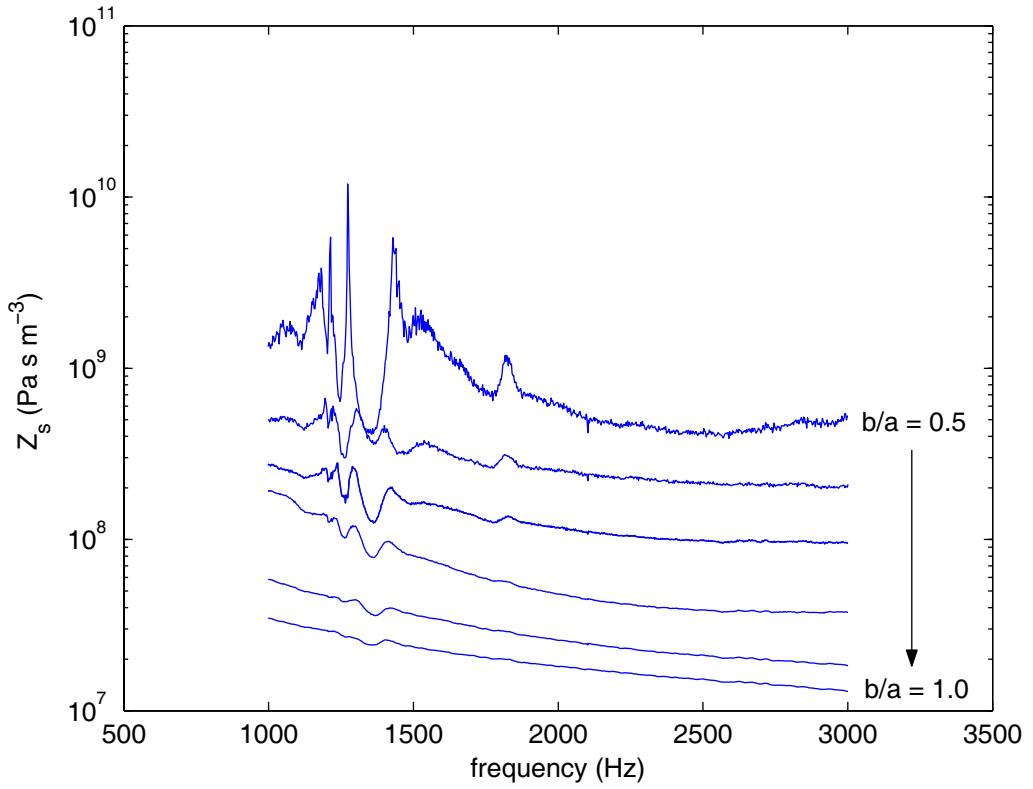


Figure 4.10: The shunt impedance Z_s for closed finger holes of length 1.0 mm and $b/a = 0.5$ to 1.0, $a = 7.5$ mm.

hole has zero compliance and the shunt impedance is infinite.) Nevertheless a clear linear trend is apparent, and there is no clear dependence on t .

The data of Figure 4.11 were fitted with the formula

$$t_{\text{finger}}/b = -0.76b/a. \quad (4.11)$$

This equation is independent of t , although for longer holes the length correction might be expected to be reduced a little, since a longer hole intersects with a larger diameter cylinder at the outside of the instrument, resulting in less finger protrusion. The fit formula (4.11) is sufficient for modelling the classical flute, and the influence of t might only be important for modelling instruments with holes meeting a nearly flat surface, as is the case for the bassoon.

4.3.2.2 Series impedance and length correction

The series length correction for closed finger holes is shown in Figure 4.12. For large holes and thin walls, $t_a^{(c)}$ is positive, suggesting that the extent of the finger protruding into the hole more than compensates for the cavity ordinarily created by a closed tone hole, and the flow narrows near the closed finger hole. A simple empirical correction to the equation of Dubos et al. (1999) (2.36) allows a reasonable fit to the experimental data of Figure 4.12. The extent of finger protrusion into the hole is quantified in relation to the length t_0 . For any given hole diameter ratio b/a , when $t = t_0$ the closed finger hole has no net effect on the flow and $t_a^{(c)} = 0$.

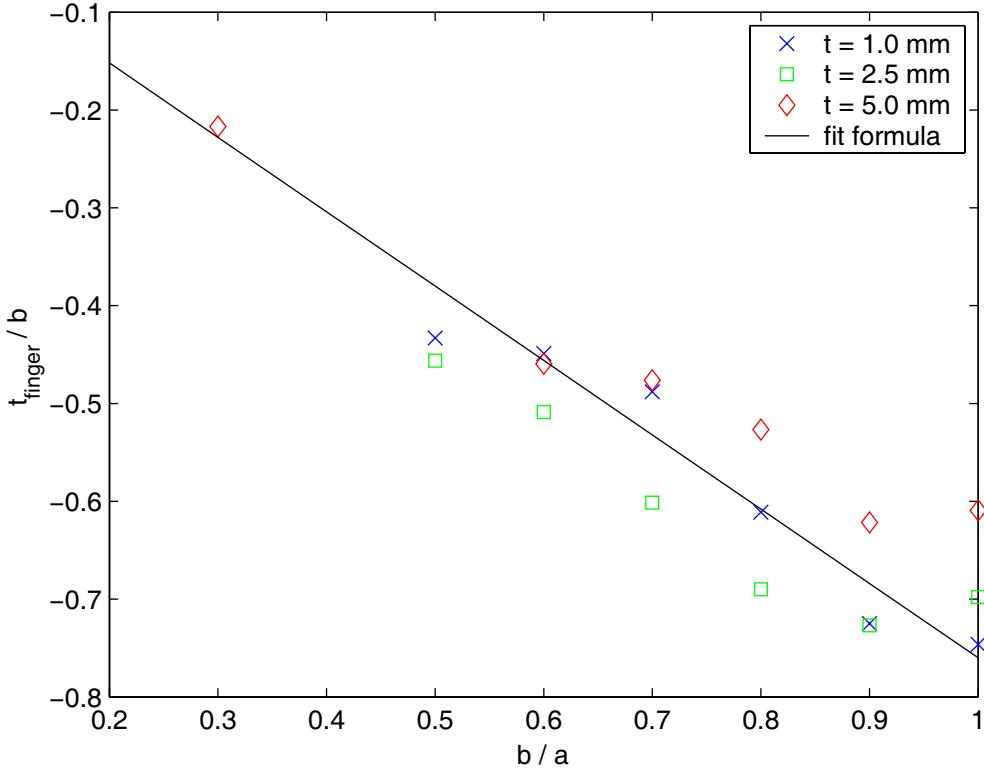


Figure 4.11: The measured length correction t_{finger} and fit formula (4.11).

For closed finger holes the following modified version of (2.36) is used:

$$t_a^{(c)} = \begin{cases} \frac{-b\gamma^2}{1.78 \coth(1.84(t-t_0)/b) + 0.940 + 0.540\gamma + 0.285\gamma^2} & t \geq t_0 \\ \frac{-b\gamma^2}{1.78/(1.84(t-t_0)/b) + 0.940 + 0.540\gamma + 0.285\gamma^2} & t < t_0, \end{cases} \quad (4.12)$$

where for the case $t < t_0$ the $\coth(x)$ function is replaced by its limit $\lim_{x \rightarrow 0} \coth(x) = \frac{1}{x}$.

The finger protrusion length t_0 was fitted empirically to the experimental data by the equation

$$t_0 = b[0.55 - 0.15 \operatorname{sech}(9t/a) + 0.4 \operatorname{sech}(6.5t/a)(\gamma - 1)]. \quad (4.13)$$

Equation (4.12) is plotted along with the experimental data in Figure 4.12.

4.4 CONCLUSIONS AND FURTHER DIRECTIONS

In this chapter the reactive impedance components in the equivalent T-circuit for open and closed finger holes were measured and compared with fit-formulae in the literature. For open finger holes existing fit-formulae were found to be sufficient, but for closed finger holes new fit-formulae were calculated in order to match better the experimental results.

Several refinements to the technique used in this chapter are expected to yield more accurate results. The dynamic range of the system is in part limited by geometrical differences among the PVC pipes used. For example, small changes in pipe diameter may swamp the flow-widening effect of a small finger hole. In further work the test pipes should be more accurately machined. Alternatively, a finger hole may be formed in a PVC test pipe and subsequently filled

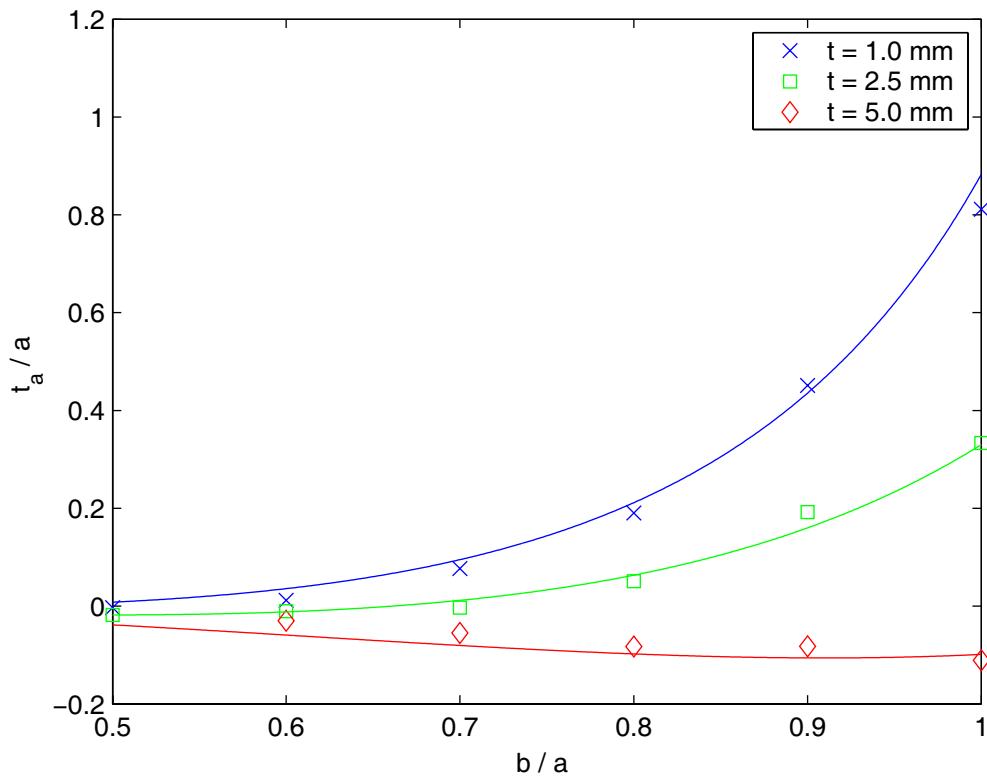


Figure 4.12: The series length correction t_a for closed finger holes and fit formula (4.12).

with a removable plug of resin. (The plug would then be reamed smooth to match the bore.) The measurement system could then be calibrated with the plug in place and then the hole could be measured after removal of the plug. This procedure possesses the additional advantage that there is no need to dismantle the apparatus between calibration and measurement.

Due to some systematic errors in the measured data no attempt was made in this chapter to quantify the resistive components of the T-junction impedances. With more careful measurements it may be possible to measure these with the system described in this chapter.

Chapter V

Impedance spectra of the flute and clarinet*

5.1 INTRODUCTION

Several authors have measured the input impedance spectra of brass and woodwind instruments (Backus 1974, 1976, Elliott et al. 1982, Caussé et al. 1984, Wolfe et al. 2001a) but such measurements are often prone to systematic errors and, depending on the excitation signal used, may take a great deal of time to measure over a wide frequency range. In this chapter, a spectrometer using three microphones and calibrated on resonant-free loads (see Chapter 3) is used to measure the input impedance of several flutes and clarinets. Using such impedance spectra, many differences in tuning and timbre between the instruments are explained.

A database of input impedance spectra for the flute (Wolfe et al. 2001b) has been used to develop software tools that are widely used by flutists (Botros et al. 2002). In the future it may be possible to use the measured database of clarinet spectra in an analogous way. In this thesis, the measured flute impedance spectra are used to fit a semi-empirical model for flutes of arbitrary geometry. This model forms the basis of a software tool being developed for instrument makers.

5.2 MATERIALS AND METHODS

5.2.1 Instruments and fingerings

The following instruments were measured:

- modern flute (Pearl PF-661, open hole, measured with both a C foot and a B foot)
- classical flute (made by Terry McGee, Canberra, Australia)
- B \flat and A clarinets (Yamaha Custom CX).

The dimensions of these instruments are provided in the XML files `ModernFlute.xml`, `ClassicalFlute.xml` and `BbClarinet.xml` in Appendix B. The classical flute, an experimental instrument, is described in Wolfe et al. (2001a), although several keys have been added to the flute in the intervening time—it now has four keys, which when operated give the notes E, B \flat , C and G \sharp (the flute is designed to play in the key of D without the use of any keys). The two clarinets measured have the same bore diameter and differ by approximately 6% in length. In addition to the above-mentioned instruments, the modern and classical flute headjoints were both measured without the flute body. Some simple T-junctions were also measured, as mentioned in §5.3.3.

* Parts of this chapter have been published as Dickens, P., France, R., Smith, J. & Wolfe, J. (2007), 'Clarinet acoustics: introducing a compendium of impedance and sound spectra', *Acoustics Australia* 35(1), 17–24.

The position of the cork in the head joint of the modern flute was set at 17.5 mm from the centre of the embouchure hole, and the tuning slide was set at 4 mm. These are typical values used by flutists playing at standard pitch. For the classical flute the cork was set at 19 mm and the tuning slide at 12.8 mm (this was determined empirically by the maker Terry McGee as the correct position for the flute to play at A 440, although his playing style may not be typical).

All standard and many alternative and multiphonic fingerings were measured for the modern flute with both C and B foot. Classical flutes currently available differ widely in design and hence no authoritative standard exists. For the measurements in this chapter, the fingerings given in Porter (2005) were used. In all cases the fingerings are shown on each graph of the results.

The impedance of the clarinets were measured over the range from E3 to E7 (written) for all standard fingerings and a selection of multiphonic and alternate fingerings.

5.2.2 The impedance spectrometer

The details of the impedance spectrometer have been described in Chapter 3. The impedance spectra are measured between 200 Hz and 4 kHz for the flute and between 120 Hz and 4 kHz for the clarinet (a range that encompasses the fundamental frequency of all of the instruments' notes and includes their cut off frequencies).

A 7.8 mm inner diameter brass measurement head is used in addition to the 15 mm head described in Chapter 3. In all other respects this impedance head is identical to the 15 mm version. The 7.8 mm impedance head may be used to measure the input impedance of flutes and clarinets as the outlet fits within the embouchure hole of a flute and the mouthpiece of a clarinet. (The cross-sectional area of this head is comparable with the area through which air flows between clarinet and mouthpiece, i.e. the gap between reed and mouthpiece. It is also comparable with the area of the hole at the embouchure of a flute when played.)

The impedance head is calibrated as described in Chapter 3 using a solid stop (infinite impedance load) and a 7.8 mm semi-infinite pipe.

5.2.3 Attachment of instruments

The flutes were coupled to the impedance head as in Wolfe et al. (2001a) (see also Smith et al. 1997) with a concave attachment matching the curvature of the embouchure striker plate (Figure 5.1 (a)). The thickness of the concave attachment was accounted for in calibration, so that the reference plane for impedance measurements was situated at the embouchure hole. For the spectra presented in this chapter and in Appendix A the impedance of a stub of length 5 mm and diameter 7.8 mm was added to the measured impedance to account (at least approximately) for the radiation impedance baffled by the player's face and lips (see Wolfe et al. 2001a). The impedance was thus measured upstream of a discontinuity, since the diameter of the embouchure hole on the outside of the instrument is greater than 7.8 mm for both head-joints measured. This measured impedance will be different to the impedance that would be measured by an impedance head with diameter matched to the input radius of the instrument. However, in the played instrument a similarly sized discontinuity exists, since the player's lips occlude the embouchure hole to a significant extent.

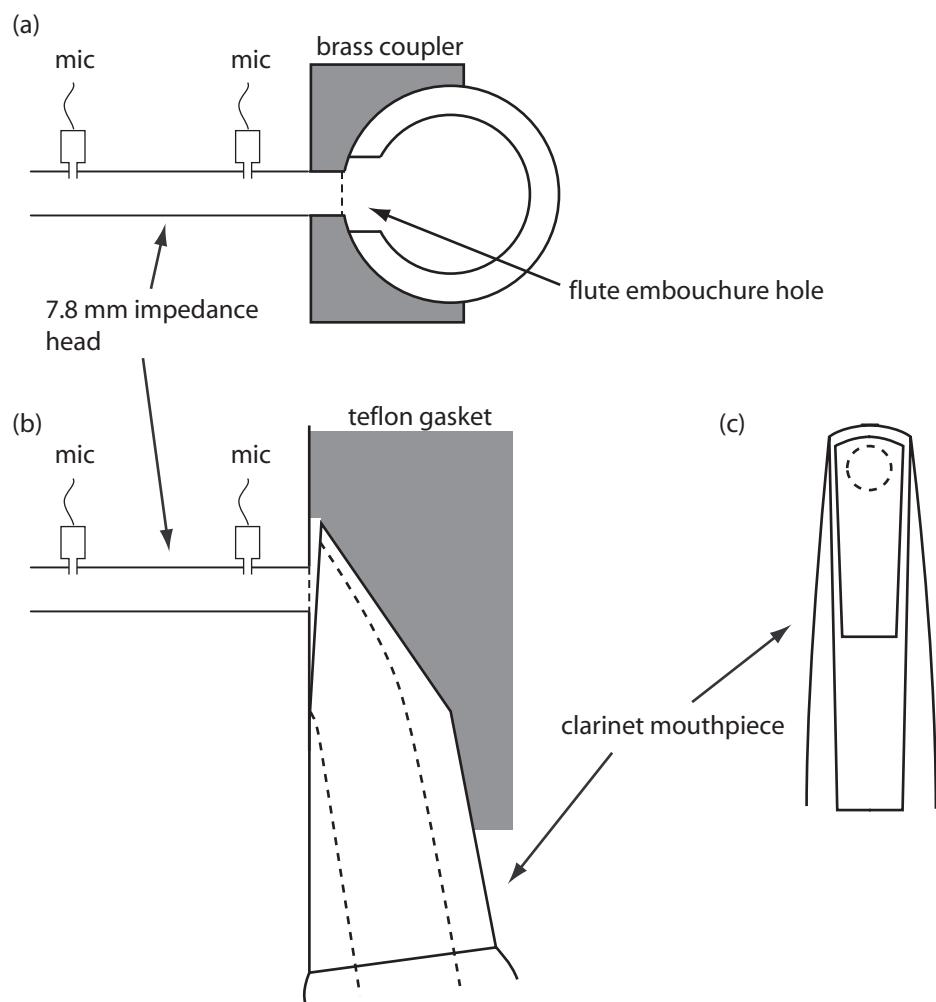


Figure 5.1: Sketch showing the attachment of the flute and the clarinet to the impedance head. (a) Side view of the flute showing impedance head, brass coupler and embouchure hole. (b) Side view of the clarinet showing impedance head, teflon gasket and mouthpiece. (c) Top view of the clarinet mouthpiece showing placement of the impedance head output (dotted) relative to the mouthpiece lip. In (a) and (b) the reference plane for impedance measurements is shown as a dotted line perpendicular to the impedance head.

As shown in Figure 5.1 (b) and (c), the measurement head is attached to the clarinet mouthpiece with the outlet positioned almost touching the top lip of the mouthpiece (preliminary measurements have shown that this positioning is not critical). A teflon gasket prevents any leaks. Positioned thus, a volume of $60 \mu\text{L}$ is added to the volume of air in the clarinet closed by the reed. However, according to Nederveen (1998), the compliance of a number 4 reed (reasonably hard) is equal to the compliance of 1.1 mL of air. Therefore the compliance of $1040 \mu\text{L}$ of air is added algebraically in parallel to the measured impedance, yielding the impedance of the clarinet with reed, a much more useful quantity for determining playing characteristics. The spectra presented in Appendix A have all been thus corrected for the added volume and the reed compliance.

5.2.4 Measurement conditions

The flute measurements were made at $T = 21.5 \pm 0.3^\circ\text{C}$ and relative humidity $38 \pm 1\%$. The clarinet measurements were made at $T = 27.5 \pm 0.5^\circ\text{C}$ and relative humidity $61 \pm 1\%$.

5.3 RESULTS AND DISCUSSION

5.3.1 Flute and clarinet impedance spectra compared

A modern flute and a clarinet are, to zeroth order, similar geometrically, i.e. if one neglects the bell of the clarinet and the variations in bore. Yet they differ significantly in the range and timbre of sounds they produce. The instruments are of approximately equal length and the bores of each are mostly cylindrical. The difference, of course, is that the clarinet is almost closed by the reed at its embouchure (i.e. it is almost a closed pipe) whereas the flute is open to the air at its embouchure (almost an open pipe). The lowest note on a flute is almost an octave higher than the lowest note on a clarinet, the flute overblows an octave while the clarinet overblows a twelfth and their sound spectra reflect this difference, at least for notes in the lowest register. (The principal differences between the flute and the clarinet are due to the different excitation mechanisms. Some—but only some—of these differences are summarised in the low order approximation in which one is approximated by an open pipe, and the other by a closed pipe. These approximations, of course, omit virtually all details of air jets, upon which flutists expend hours and years of practice, and the mechanical properties of reeds, which are of passionate interest to clarinettists. Nevertheless, it is interesting to compare and to contrast the acoustics of idealised open and closed pipes in terms of their acoustic impedance spectra and to compare these with the rather more complicated behaviour of the bores of flutes and clarinets.)

Figure 5.2 shows an impedance spectrum measured using the 15 mm impedance head for an open pipe of length 650 mm and diameter 15 mm. The broad features of this spectrum are similar to the spectra for a modern flute and a modern clarinet, both with all finger holes closed. The primary difference between the instruments is the condition at the embouchure. Flutes are open at the embouchure, and hence operate at impedance minima. Clarinets operate at impedance maxima. The first six resonances and nine antiresonances of this pipe are shown in musical notation in Figure 5.2. While the notes shown for a flute are approximately correct, orchestral clarinets play somewhat sharper than the notes shown here for a simple cylindrical pipe (due primarily to the effect of the bell).

The impedance spectra of real flutes and clarinets are clearly not identical to the spectrum of an open pipe of equivalent length. The Z spectra of the flute fingered to play C5 and the B \flat clarinet fingered to play C4 are shown in Figure 5.3, along with the impedance of a 15 mm diameter pipe of length 325 mm (approximately equal to the equivalent length of both instruments).

The juxtaposition of these three curves allows us to make explicit comparisons that show interesting features of these instruments. For the clarinet, the spectrum is reasonably similar to that of the cylindrical pipe for frequencies less than about 1.5 kHz, and the spacing between impedance maxima (or minima) is about $\frac{2c}{L}$, where L is the effective length of the cylindrical pipe

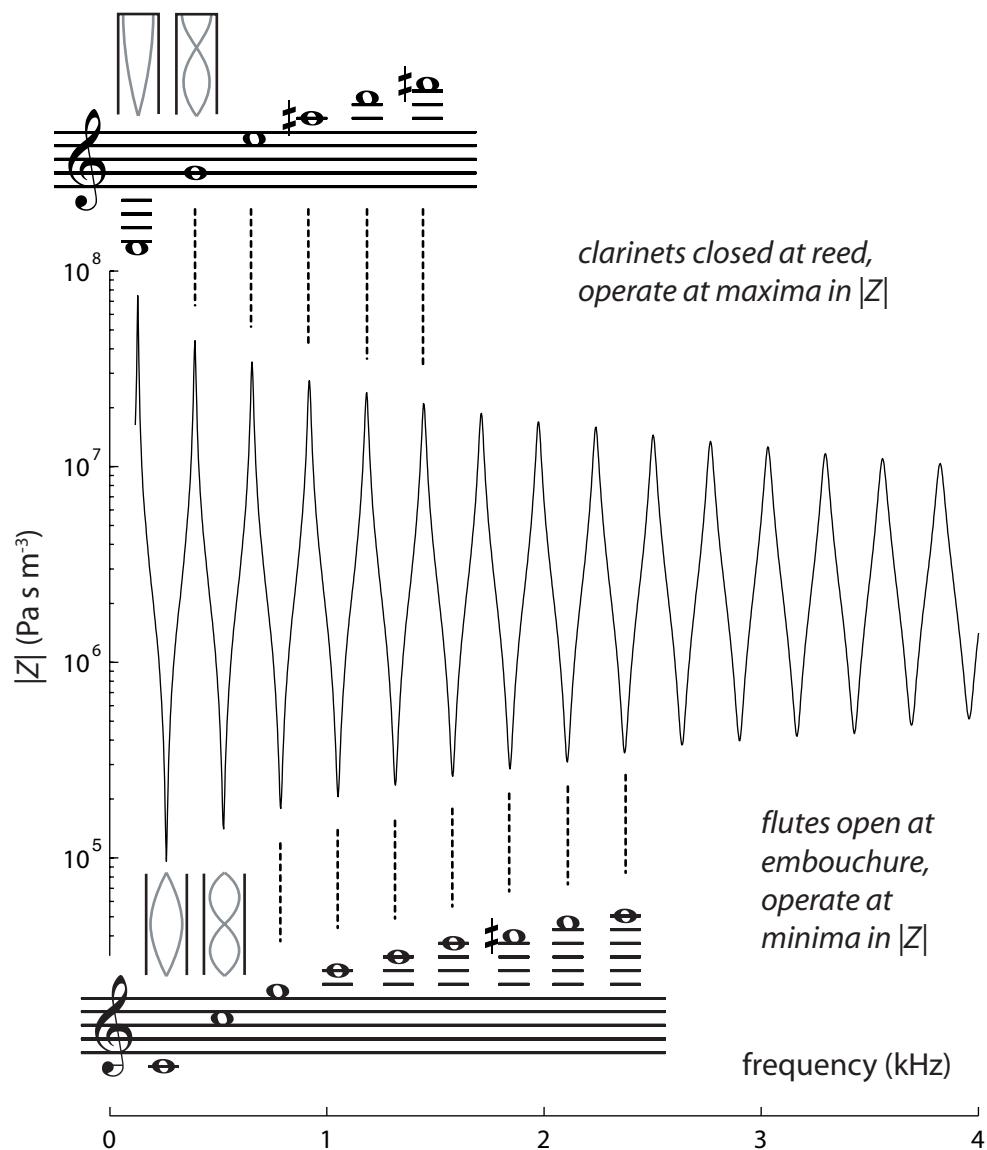


Figure 5.2: The measured impedance spectrum for an open pipe of length 650 mm and diameter 15 mm. A flute with all holes closed plays the notes shown *below* the spectrum. A clarinet with all holes closed plays close to the notes shown *above* the spectrum.

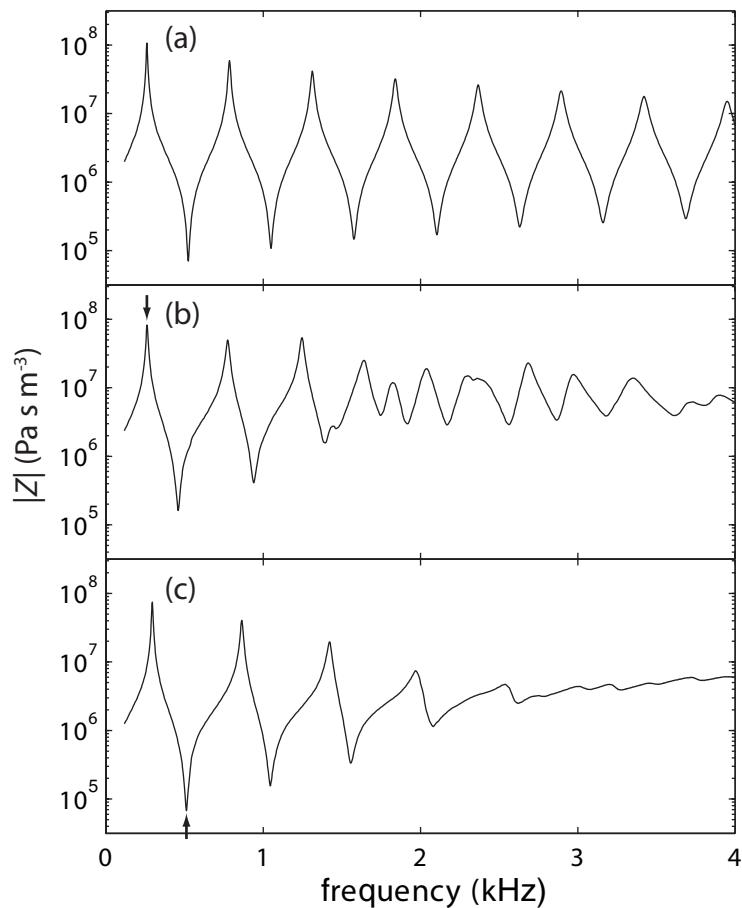


Figure 5.3: Measured impedance spectra of (a) an open pipe (15 mm diameter, 325 mm long), (b) a B \flat clarinet fingered to play C4 (written D4) and (c) a flute fingered to play C5. Arrows in (b) and (c) show the extrema corresponding to the played notes.

and, for the clarinet, approximately the length from mouthpiece to the first of the array of open tone holes. The cut off frequency for this array of tone holes (Benade 1976) is approximately 1.5 kHz. At frequencies well above this, the inertia of the air in the open holes is sufficient to isolate acoustically the wave in the bore from the radiation field outside. Consequently, at frequencies well above this, the frequency spacing between maxima is much smaller: approximately $\frac{2c}{L_{\text{clar}}}$, where L_{clar} is approximately the length of the whole clarinet (with some correction for the bell). A consequence is that a note played with this fingering (C4) has impedance maxima approximately corresponding to the first, third, fifth and seventh harmonic, but there is no systematic pattern relating higher resonances to harmonics. Further, a note using a similar fingering in the next register (G5), whose fundamental involves the second maximum, has no systematic relation between higher harmonics and impedance maxima. This explains why the common oversimplification that clarinet notes have strong odd harmonics is very limited: it applies primarily to notes in the lowest register, and then only for harmonics falling below or near the cut off frequency.

A similar effect occurs in the flute, whose cut off frequency is about 2 kHz, but it is clearly visible only at much higher frequencies (Wolfe & Smith 2003). The reason is that the embouchure of the flute has a Helmholtz resonator, formed by the enclosed volume of air between the embouchure and the cork (the compliance) and the air in the embouchure hole (the mass). Near its rather broad resonance (centred at about 3 kHz), this Helmholtz resonator acts as an acoustical short circuit in parallel with the bore, which explains the lack of structure in the impedance spectrum in this frequency range.

Using such impedance spectra, it is possible to explain not only such general features, but also some of the particularities of individual fingerings. Brief examples of such discussions are given for many of the results on the web site of the Music Acoustics Laboratory at the University of New South Wales <<http://www.phys.unsw.edu.au/music>>.

5.3.2 Clarinet impedance spectra

Figure 5.4 shows the magnitude and phase of the impedance spectra (with added reed compliance) for the lowest note (written E3) on the B♭ clarinet. Both measurements were made using an output waveform with 2^{15} points. 32 cycles were averaged to reduce noise. Spectra similar to that shown in Figure 5.4 were measured for all standard and many non-standard clarinet fingerings. The resulting database is available at <<http://www.phys.unsw.edu.au/music/clarinet/index.html>> and shows for each measured fingering the magnitude of the impedance spectrum along with a sound spectrum (the clarinet was played by Catherine Young) and fingering schematic. A brief commentary on each page explains some interesting features of the note, and any relationship with the impedance spectrum. Figure 5.5 is an image from the above-mentioned web site showing the magnitude of the impedance spectrum and a fingering diagram for the clarinet fingered to play F5.

5.3.3 Modelling the flute headjoint

A flute headjoint has a cylindrical or tapered bore, closed at one end by the cork and open where it joins the flute body. A single hole (the ‘embouchure hole’) near the corked end is applied to

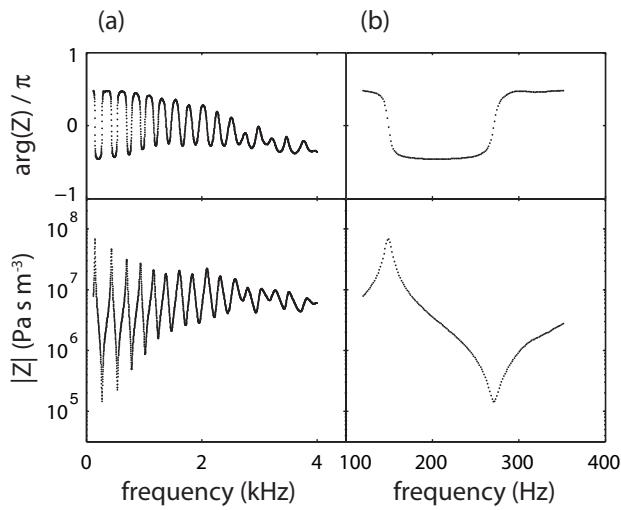


Figure 5.4: The input impedance of the B \flat clarinet fingered to play E3 (written). Each data point corresponds to a measurement. In (a) an output spectrum is used comprised of frequency components from 120 Hz to 4 kHz, while in (b), which shows the first two extrema in detail, only frequency components from 120 Hz to 350 Hz are included.

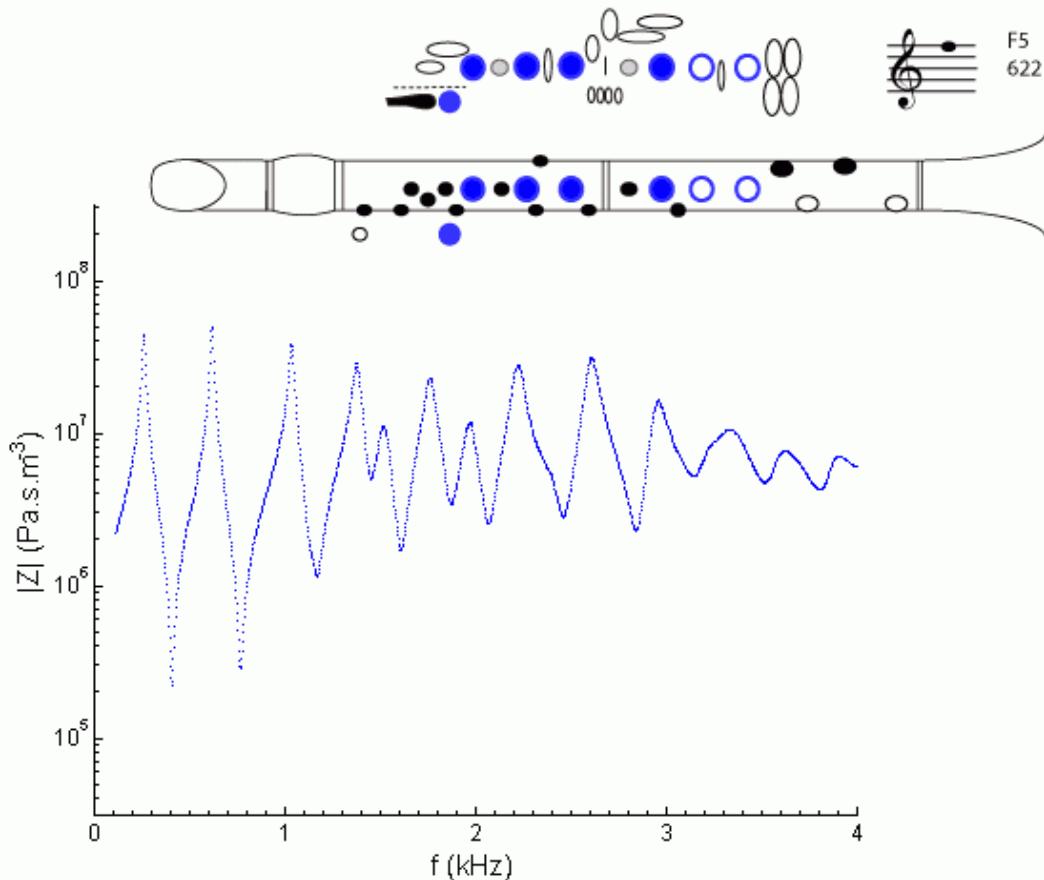


Figure 5.5: The magnitude of the impedance spectrum and fingering diagram for the B \flat clarinet playing F5.

the player's embouchure. The embouchure hole is usually slightly elliptical or oval in cross-section and is tapered so that it is larger in cross-sectional area closer to the bore. The modern flute measured in this chapter has a headjoint with a bore 168.0 mm long. The tuning slide section of the headjoint is cylindrical, 37.0 mm long with a bore diameter of 19.0 mm. The bore tapers smoothly towards the cork, where the diameter is 16.8 mm. The McGee classical flute headjoint has a cylindrical bore, 158.5 mm long with a bore diameter of 19.0 mm.

Acoustically, the headjoint thus comprises the embouchure hole, an upstream bore section stopped by the cork and a downstream bore section which attaches to the body of the flute. Despite such geometrical simplicity, the headjoint is arguably the most complex section of the flute to model, and may require the use of empirical parameters for a good fit.

The waveguide model was first tested on a simple T-junction, shown schematically in Figure 5.6 (for details of the model implementation, see Chapter 8). The input arm to the junction was of diameter 7.8 mm, equal to the diameter of the small bore impedance head. The T-junction was attached to the impedance head and the impedance was measured both with the output arm open and closed. The microphones are thus positioned sufficiently distant from the junction to measure only plane waves, and there is no diameter discontinuity at the reference plane of the impedance head. In a real headjoint there are two diameter discontinuities—one where the embouchure hole attaches to the impedance head and another at the junction between the embouchure hole chimney and the bore of the instrument—and these discontinuities are close enough that the evanescent modes at each can interact. This simple T-junction should be a little easier to model than a real headjoint. The measured impedance spectra along with the predictions of the model are shown in Figures 5.7 and 5.8. The prediction of the model is quite good for this simple T-junction, although the attenuation factor at high frequencies appears to be a little higher than in the model.

The input impedance spectra of the modern and classical flute headjoints were measured in order to test and refine the model. The headjoints were measured both open to the air and closed by a brass stop. The measured spectra for the open headjoints are shown in Figures 5.9 and 5.10, along with the predictions of the model, using published tone hole length corrections at the junction with the bore ((2.34), (2.35) and (2.39) in Chapter 2). The prediction of the model is quite good at impedance maxima but not at impedance minima (which occur at a higher frequency and are lower in impedance than measured). The model deviates similarly from experiment for closed headjoints (Figures 5.11 and 5.12).

Clearly, the discontinuity created at the input to the instrument by the measuring system changes the measured input impedance of the headjoint. This could in principle be accounted for using multi-modal theory, although this approach is complicated by the consideration that higher modes at the inside and outside of the embouchure hole interact. A simpler approach was found to be sufficient.

The model deviates from experiment most severely at impedance minima, when flow into the embouchure hole is greatest. However, the flow does not follow closely the walls of the embouchure hole because of the input discontinuity. If the embouchure hole is modelled as a cone with entry diameter equal to the diameter of the impedance head and exit diameter as

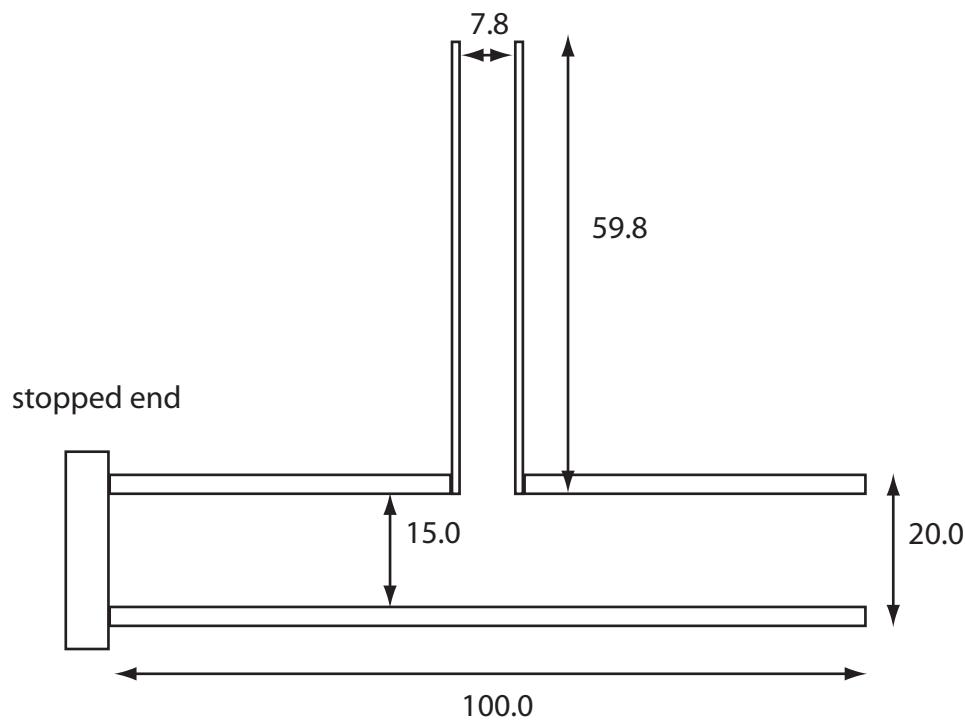


Figure 5.6: Diagram of the T-junction used to test the model predictions on a simplified head-joint. Dimensions are shown in millimetres.

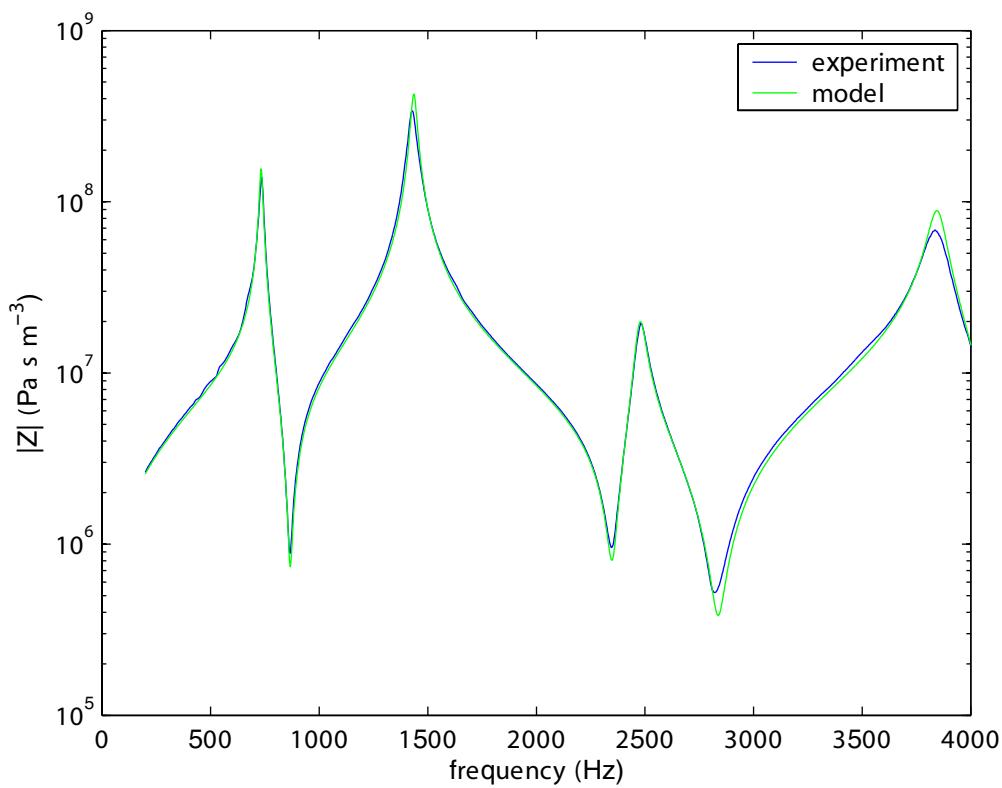


Figure 5.7: Impedance spectra (experiment and model) for the open T-junction.

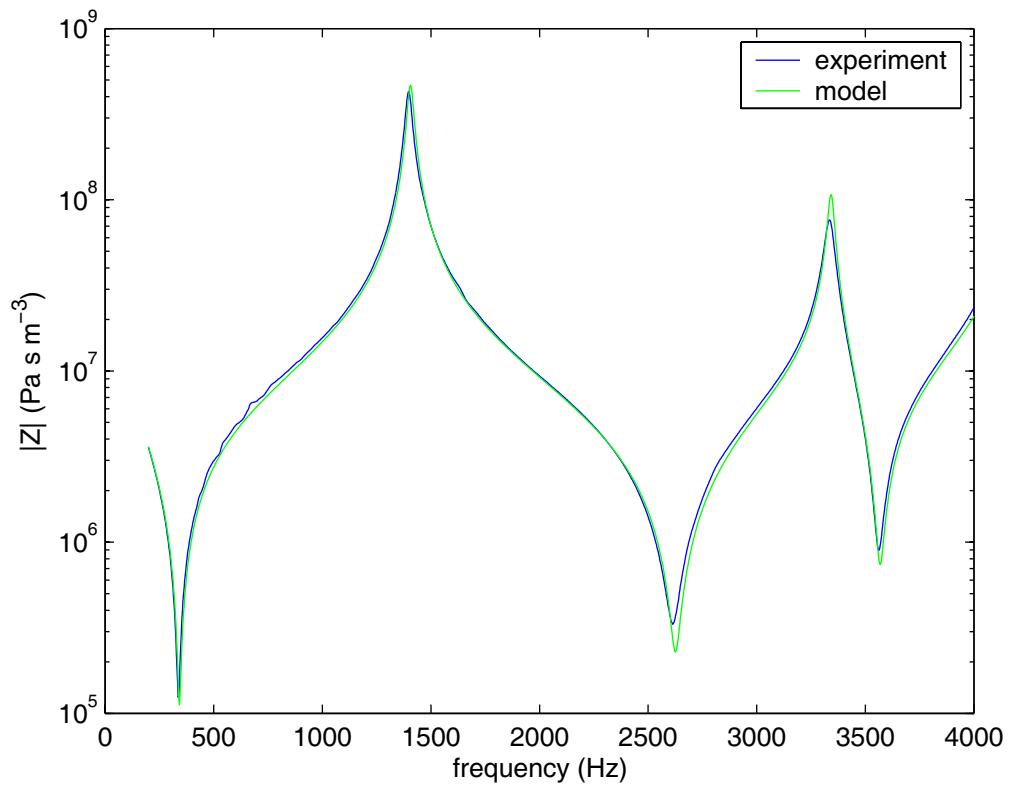


Figure 5.8: Impedance spectra (experiment and model) for the closed T-junction.

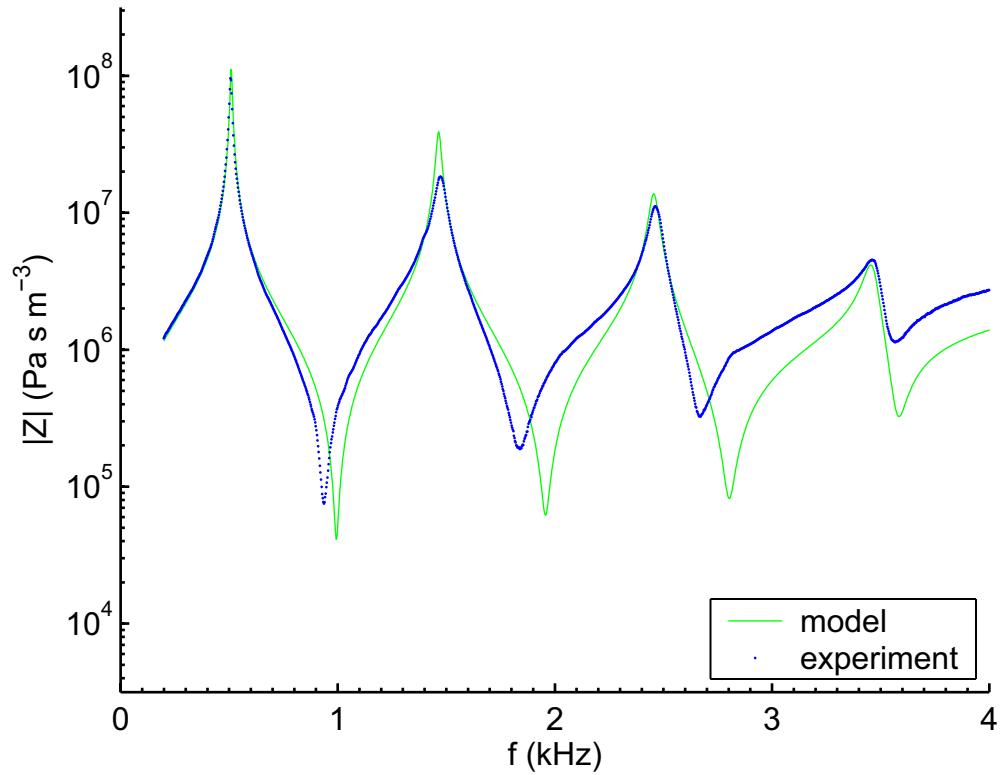


Figure 5.9: Impedance spectra (experiment and model) for an open modern headjoint before adjustment of parameters.

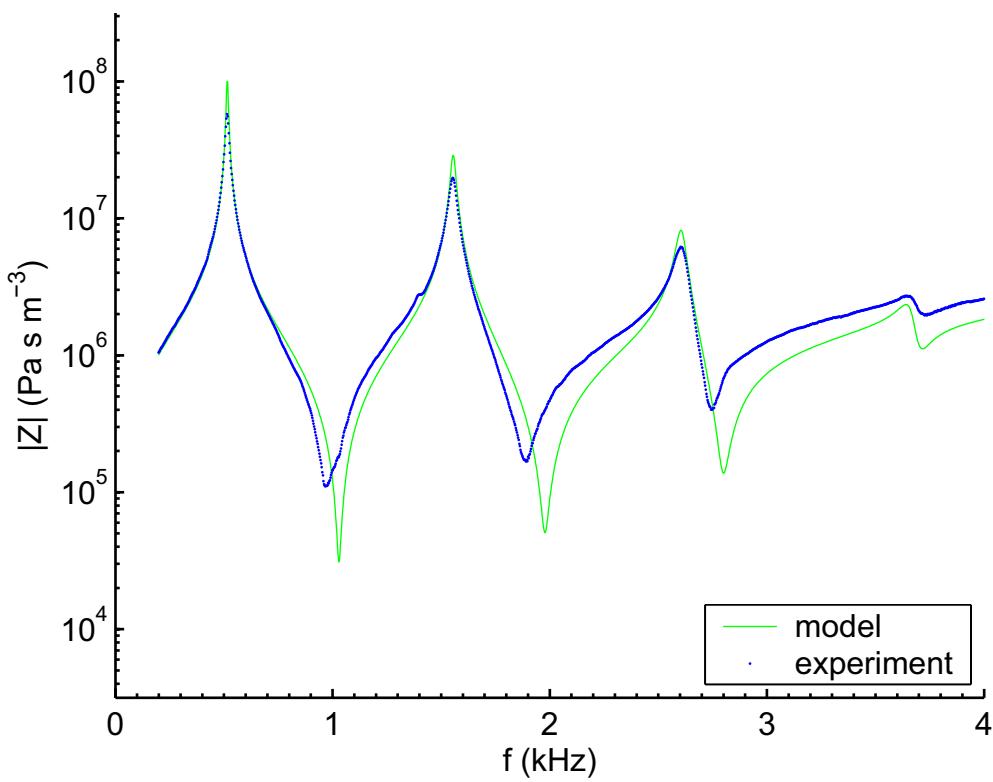


Figure 5.10: Impedance spectra (experiment and model) for an open classical headjoint before adjustment of parameters.

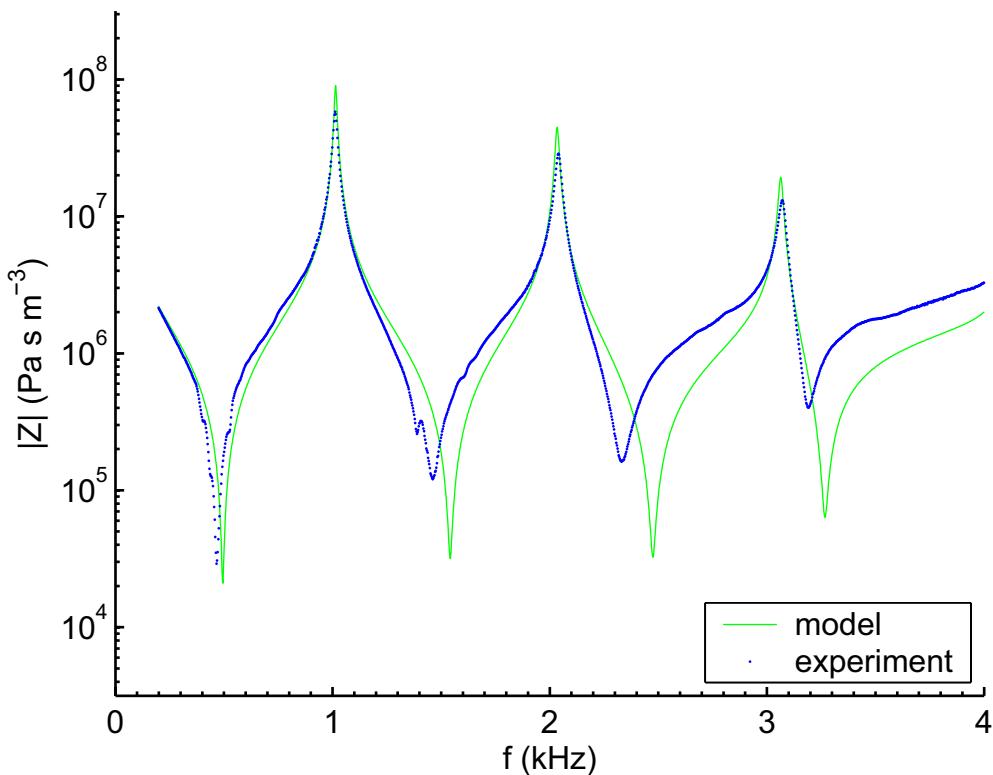


Figure 5.11: Impedance spectra (experiment and model) for a closed modern headjoint before adjustment of parameters.

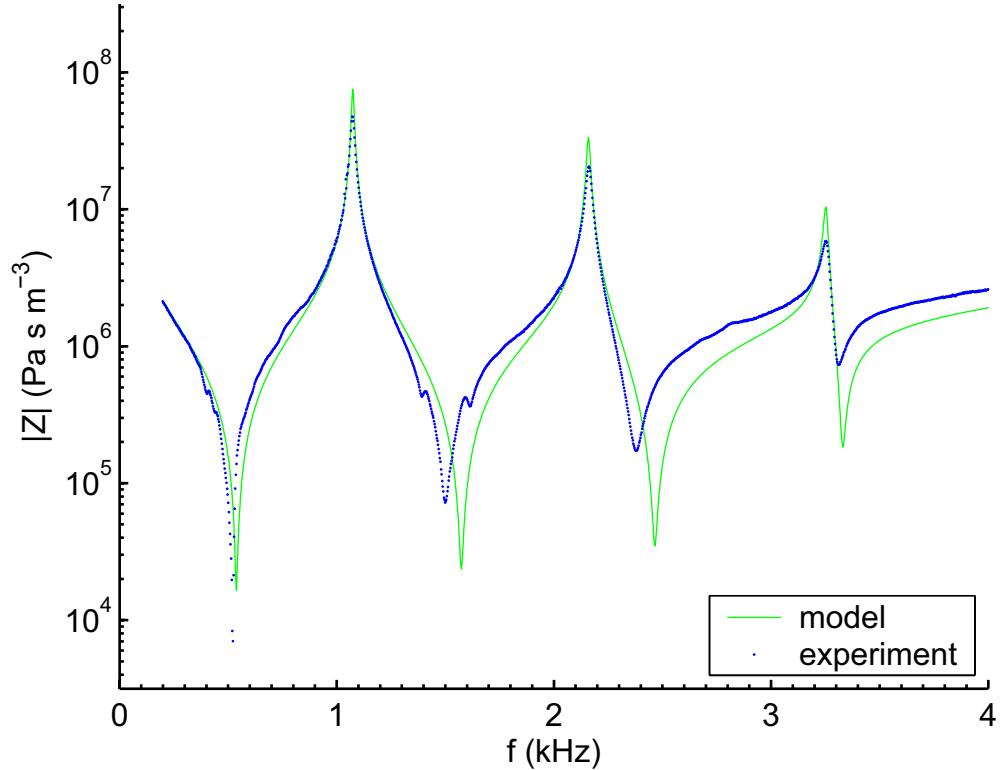


Figure 5.12: Impedance spectra (experiment and model) for a closed classical headjoint before adjustment of parameters.

measured, the model is a much closer fit to experiment, for both the modern and classical flute headjoints.

Several small refinements were made to the model to improve the fit. A small length correction was applied to the embouchure hole to achieve closer fitting of impedance minima. The correction

$$t_{\text{emb}} = 0.5\gamma^2 b, \quad (5.1)$$

where b is the inside radius of the embouchure hole and $\gamma = b/a$ for bore radius at the embouchure hole a , was found to fit both headjoints sufficiently. The necessity of such a length correction is not unexpected, given that the effective cone angle for the embouchure hole (accentuated as it is by the small entry diameter) is relatively large. The length corrections given in Chapter 2 and used in the model were all calculated for cylindrical holes, and cannot be applied to conical holes without modification.

With the above-mentioned corrections applied, the model overestimates the height and depth of impedance maxima and minima. For this reason a series resistance and a shunt conductance were added at the input to the network model. The value of each allowed precise fitting of the extrema. The values (in SI units)

$$R_{\text{emb}} = (6.9 \times 10^{-6} \text{ Hz}^{-1}) Z_0 f \text{ and} \quad (5.2)$$

$$G_{\text{emb}} = \frac{(1.3 \times 10^{-4} \text{ Hz}^{-1}) f}{Z_0} \quad (5.3)$$

for (respectively) the series resistance and shunt conductance were used. Again, using these

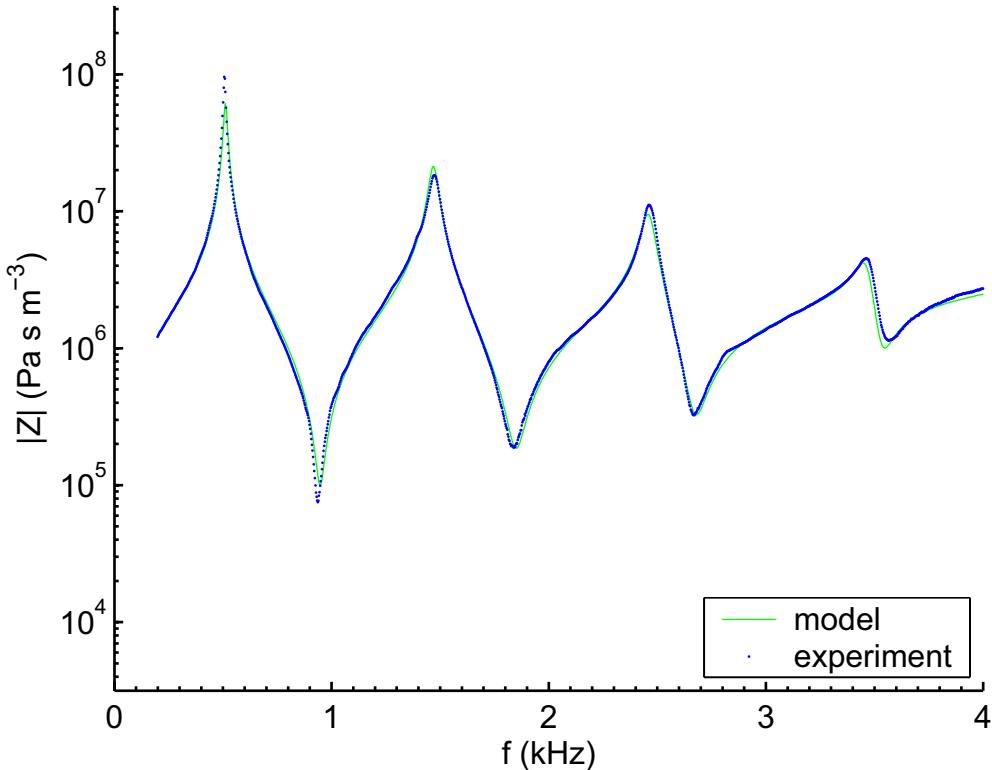


Figure 5.13: Impedance spectra (experiment and model) for an open modern headjoint after adjustment of parameters.

empirical parameters is entirely reasonable, since turbulence generated at discontinuities is a lossy process.

Experimental and modelled impedance spectra for the two measured headjoints (open and closed) are shown in Figures 5.13–5.16.

Clearly, the empirical parameters used to model the headjoint are optimised for the modern and classical flute. If a flutemaker were to design a headjoint with wildly different dimensions, then this model may not predict the input impedance accurately. Thus a more comprehensive study is in order. However, the headjoint parameters are unlikely ever to deviate greatly from these examples, since they are constrained by the player's face geometry and the physics of the air jet. For a more detailed discussion of the acoustics of flute head joints, see Benade & French (1965) and Fletcher et al. (1982).

5.3.4 Modelling the modern flute

Given an accurate model of the modern flute headjoint, we may now use the measured impedance spectra for the flute under different fingering conditions to test and refine the flute model, paying particular attention to the modelling of open and closed tone holes.

5.3.4.1 All holes closed

The modern flute with C foot plays the note C4 with all holes closed. The model was first tested on this fingering. Figure 5.17 shows the performance of a primitive model for this fingering. The headjoint model as derived in §5.3.3 was used but no account was made for the closed

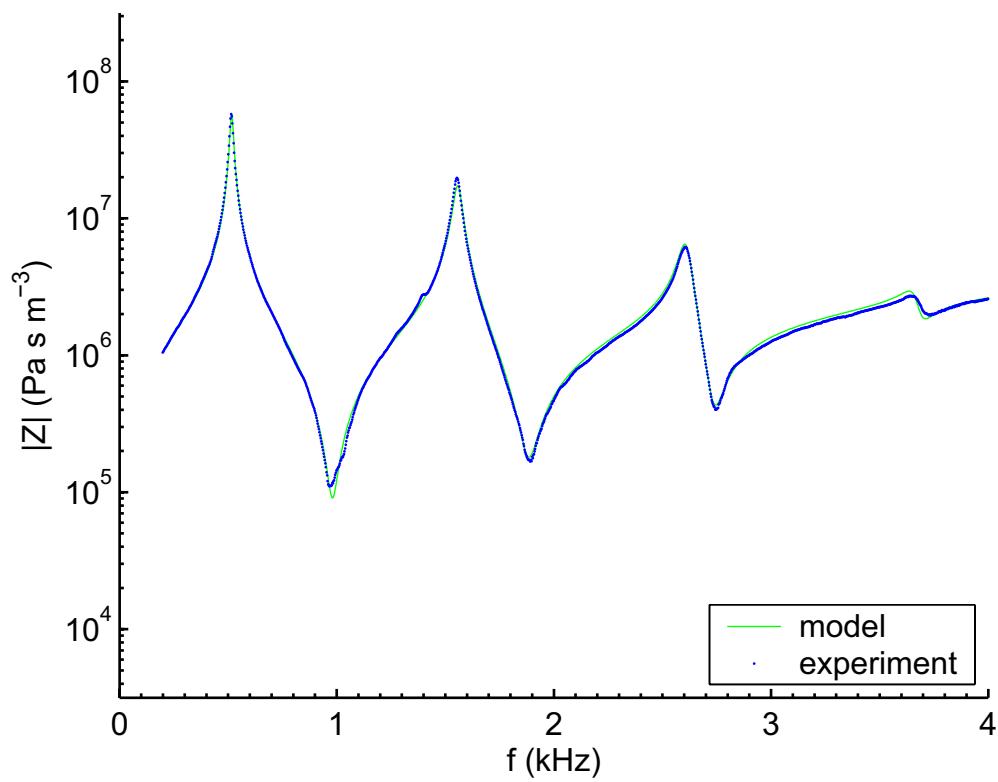


Figure 5.14: Impedance spectra (experiment and model) for an open classical headjoint after adjustment of parameters.

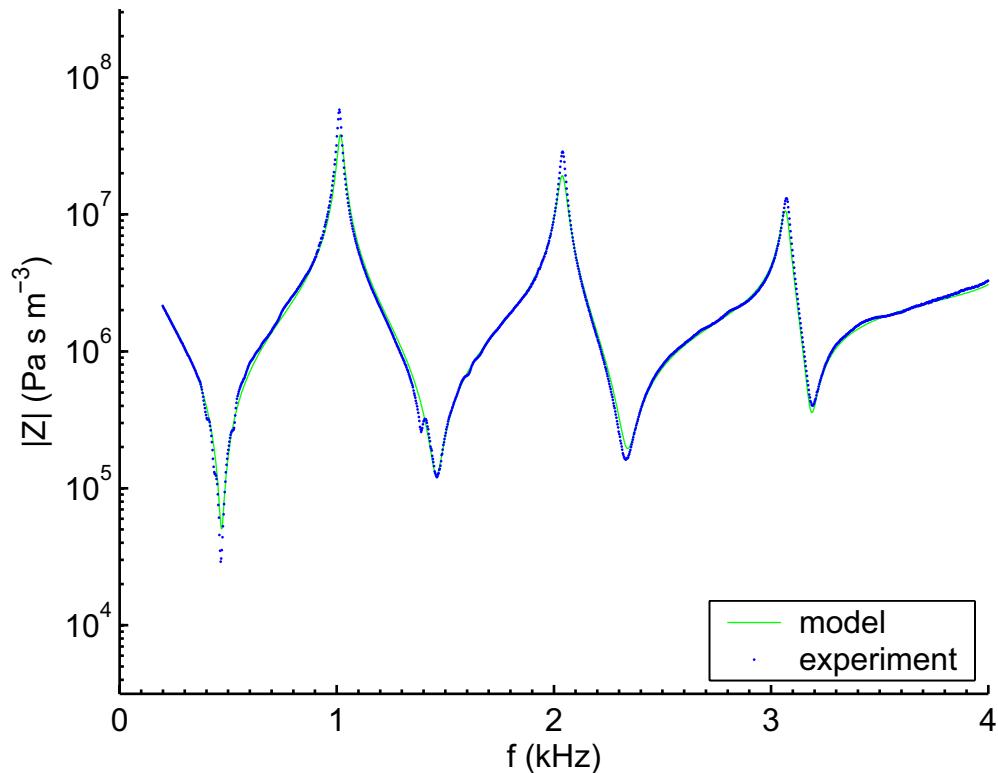


Figure 5.15: Impedance spectra (experiment and model) for a closed modern headjoint after adjustment of parameters.

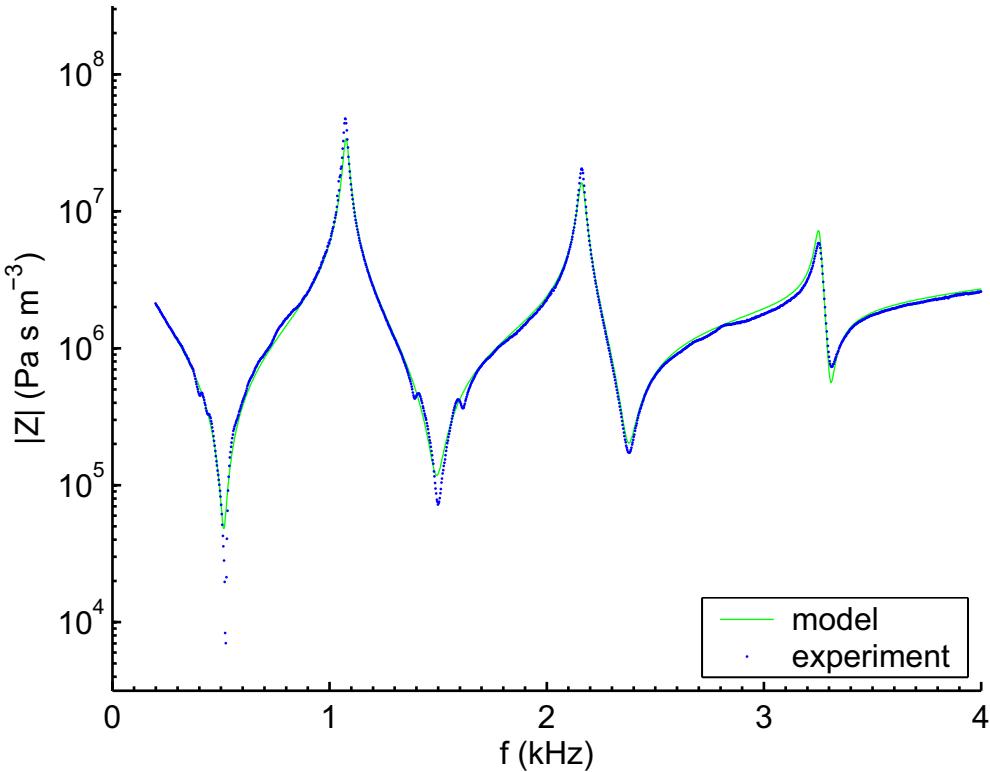


Figure 5.16: Impedance spectra (experiment and model) for a closed classical headjoint after adjustment of parameters.

tone holes—these were totally ignored in the model. The model fit in Figure 5.17 is close to the experimental data but in the model the impedance extrema are slightly higher in frequency. This is expected since a closed side hole acts mainly as a compliance (Nederveen 1998) and nearly always lowers the frequency.

Figure 5.18 shows the same experimental data with a model including the closed tone holes. The model uses the physical dimensions of each hole with the impedance corrections given in §2.2.10. Each hole is terminated by an infinite impedance. The extrema predicted by this model are slightly lower in frequency than measured. For this reason, the correction

$$t_{\text{keypad}} = -0.05b \quad (5.4)$$

was applied to the hole length (here, as in §2.2.10, b refers to the hole radius). The resulting model fit is shown in Figure 5.19. This correction is physically reasonable, since the keypad protrudes a small distance into the hole chimney and reduces the compliance of the closed hole. The impedance spectrum was also measured for the flute playing C4 with all keypad perforations plugged (not shown here). There was negligible difference between the spectra, indicating that the extra volume created by the keypad perforations and the impedance of the player's fingers are negligible in this situation.

5.3.4.2 One hole open

To test and refine the flute impedance model for a flute fingering with open holes, the fingerings C♯4 and A4 (diffuse) were tested on the modern flute with C foot. These fingerings each

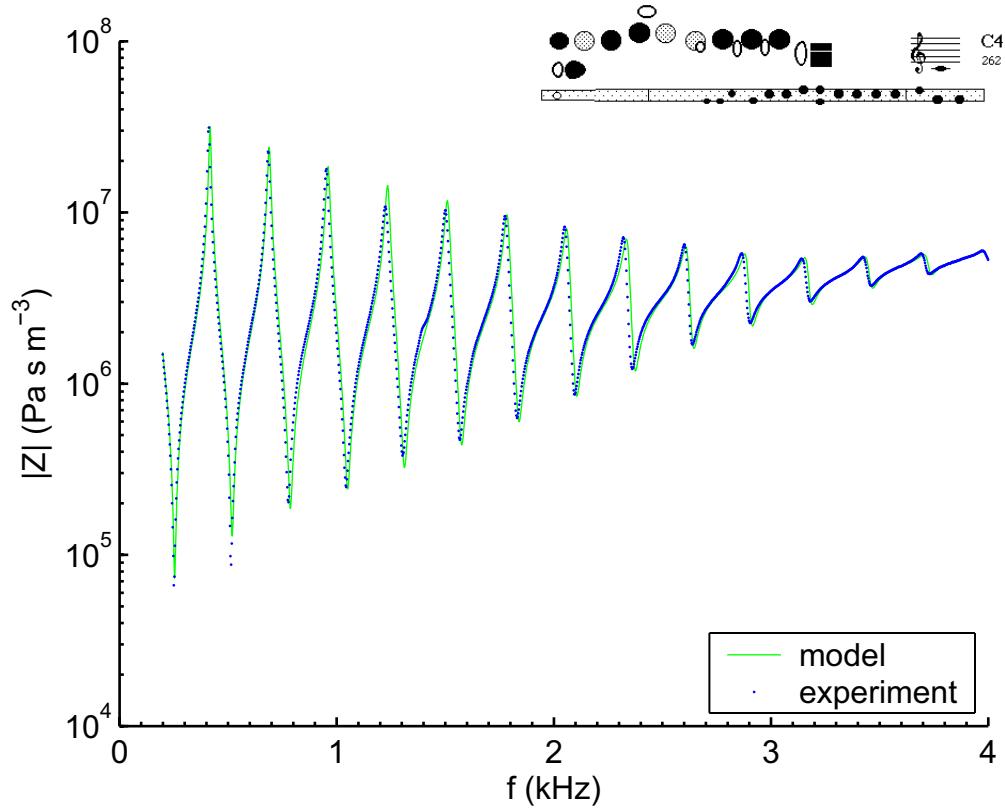


Figure 5.17: Impedance spectra (experiment and model) for the modern flute with all holes closed and the holes ignored in the model.

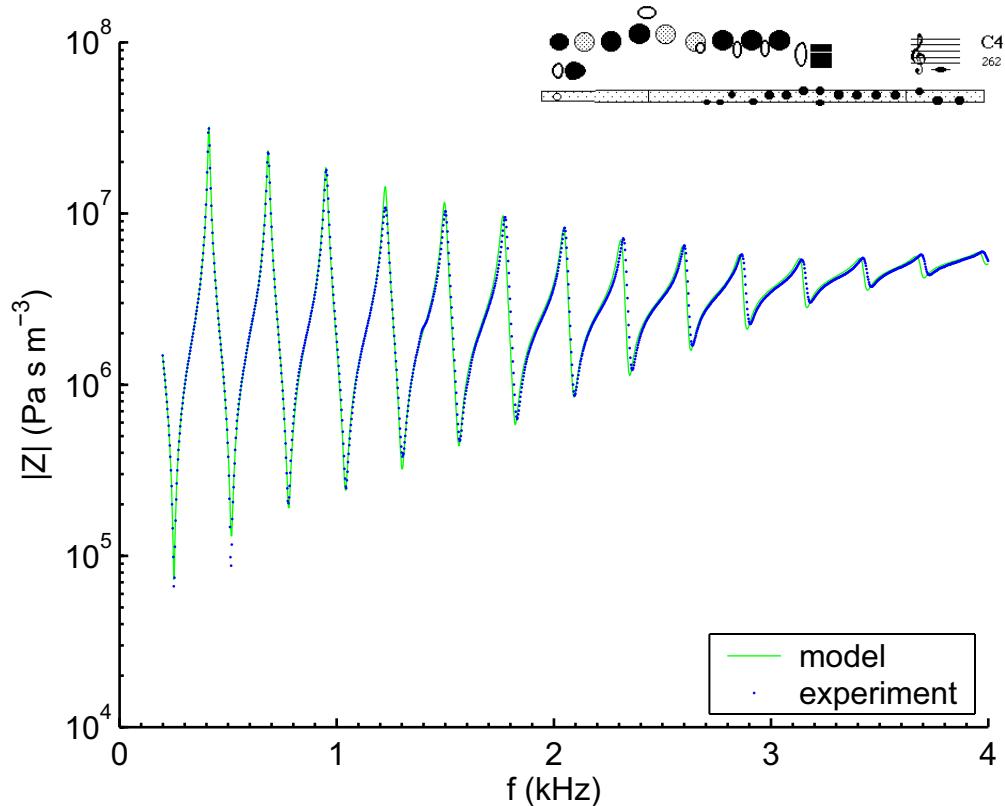


Figure 5.18: Impedance spectra (experiment and model) for the modern flute with all holes closed and without any keypad correction.

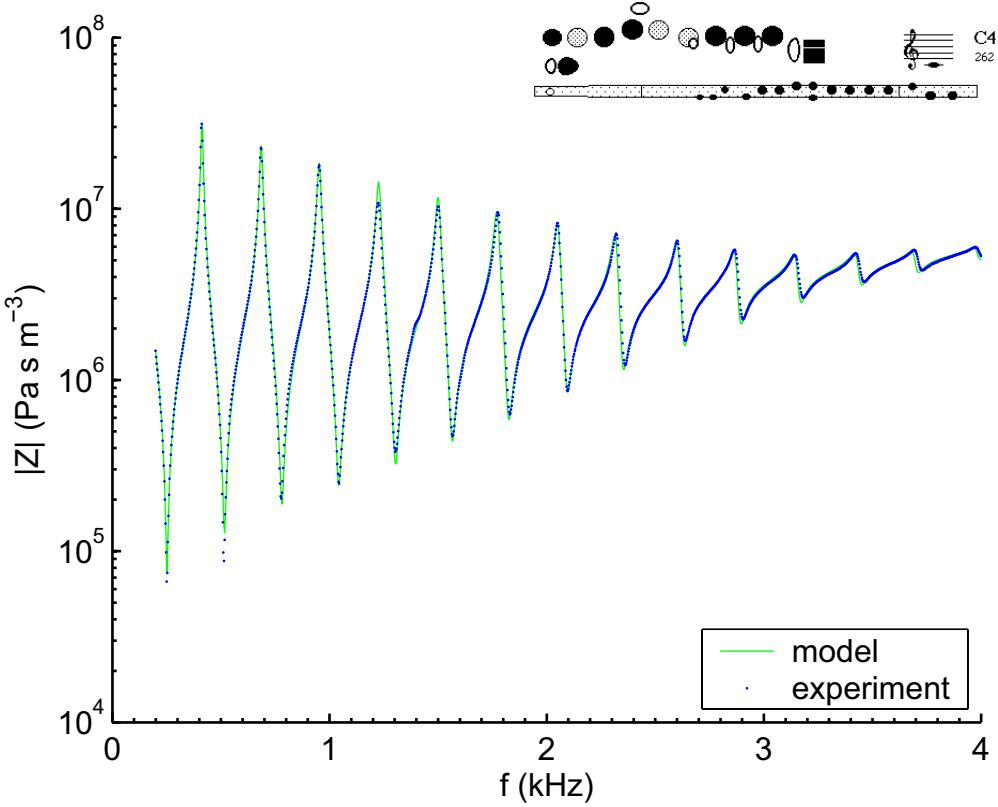


Figure 5.19: Impedance spectra (experiment and model) for the modern flute with all holes closed and with the keypad correction given in (5.4).

have one open hole and the remainder closed, and thus afford a simple test of the capability of the waveguide model to account for open holes.

Figure 5.20 compares experiment and model for the fingering C \sharp 4 on the modern flute. This fingering is identical to C4 except that the last hole is open. In Figure 5.20 the open hole has been modelled using the corrections in §2.2.10 and using a radiative load for a pipe with circular flange of width 1.0 mm (2.26). Thus any ‘shading’ of the hole by the keypad (and consequent flattening of the played note) is not yet modelled. As expected, the impedance extrema predicted by the model are sharper than measured.

The fingering A4 (diffuse) on a modern flute again has only one open hole, but in contrast to C \sharp 4 the open hole is followed by an array of closed tone holes (being roughly halfway along the flute) and the keypad over the hole is perforated (the key in question is one of the six keys normally covered by the player’s fingers). Figure 5.21 shows the prediction of the model without any shading of the hole by the raised keypad. As in Figure 5.20, the predicted impedance extrema are sharper than those measured.

Empirical fits by Dalmont et al. (2001) (equations (2.30) and (2.32)) give the extra length correction (low frequency limit) associated with a disk (which may be perforated) hanging over a hole. This equation is expected to be a reasonable approximation to the effect of a key on a flute, although in contrast to the system considered by Dalmont et al. open keys on a flute are never perpendicular to the hole axis, the surface of the key itself is compliant, and the components of the flute surrounding the hole and key may contribute to a reduction in the solid angle

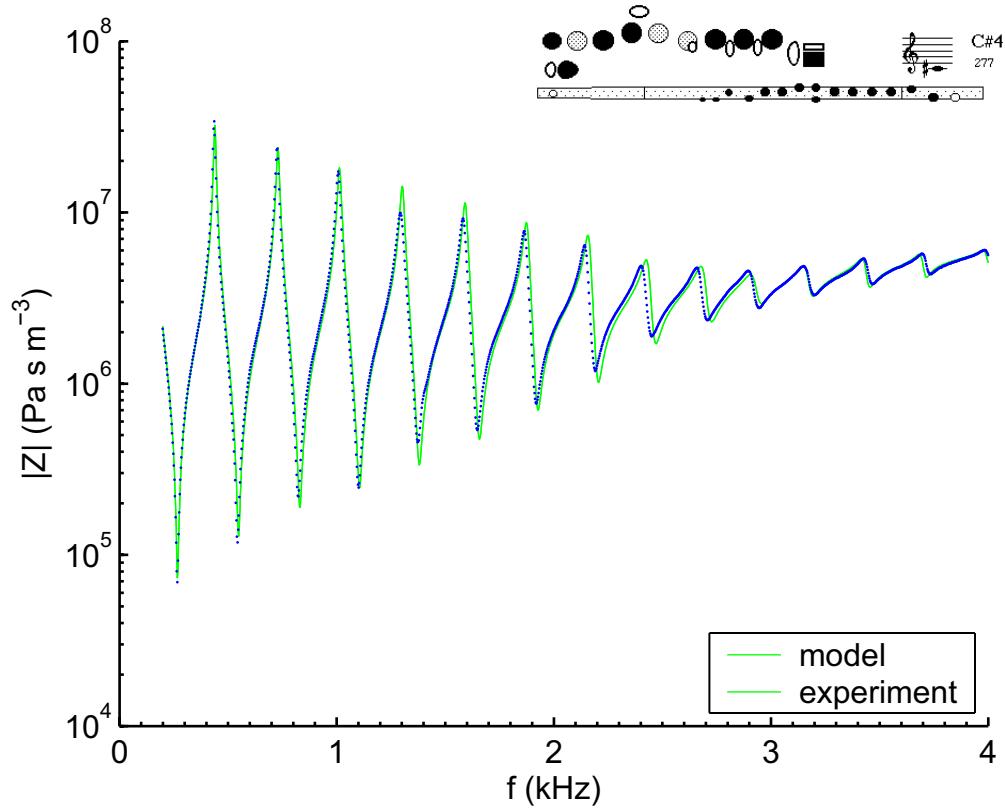


Figure 5.20: Impedance spectra (experiment and model) for the fingering C#4 on the modern flute without accounting for shading by the keypad.

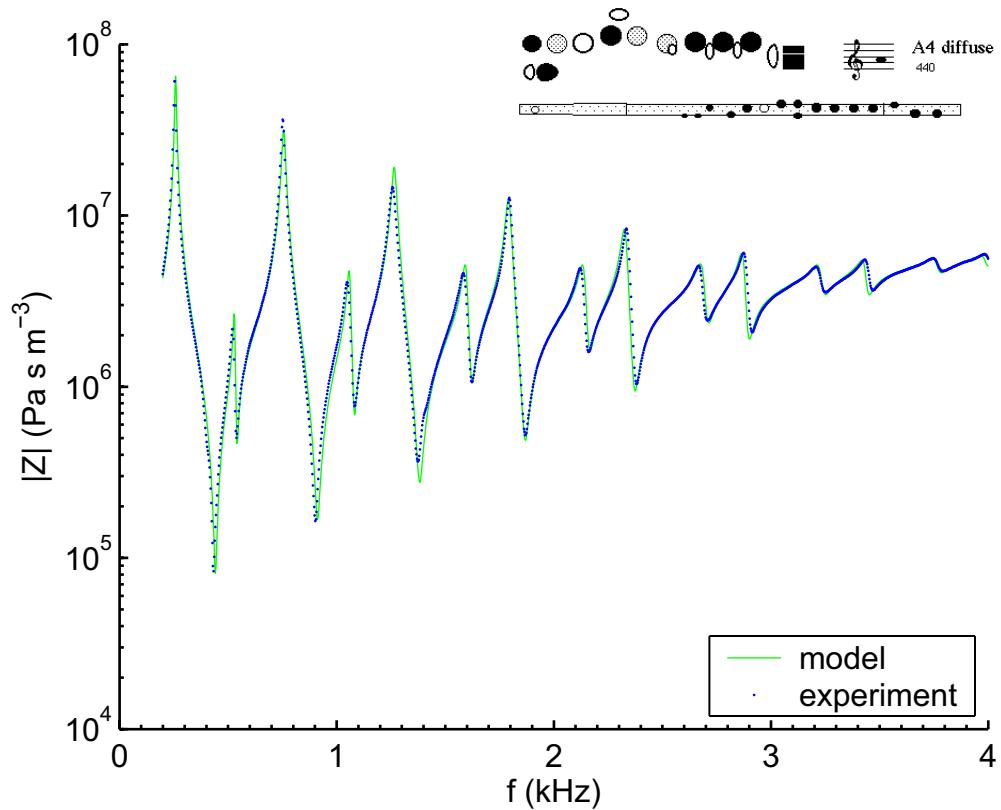


Figure 5.21: Impedance spectra (experiment and model) for the fingering A4 (diffuse) on the modern flute without accounting for shading by the keypad.

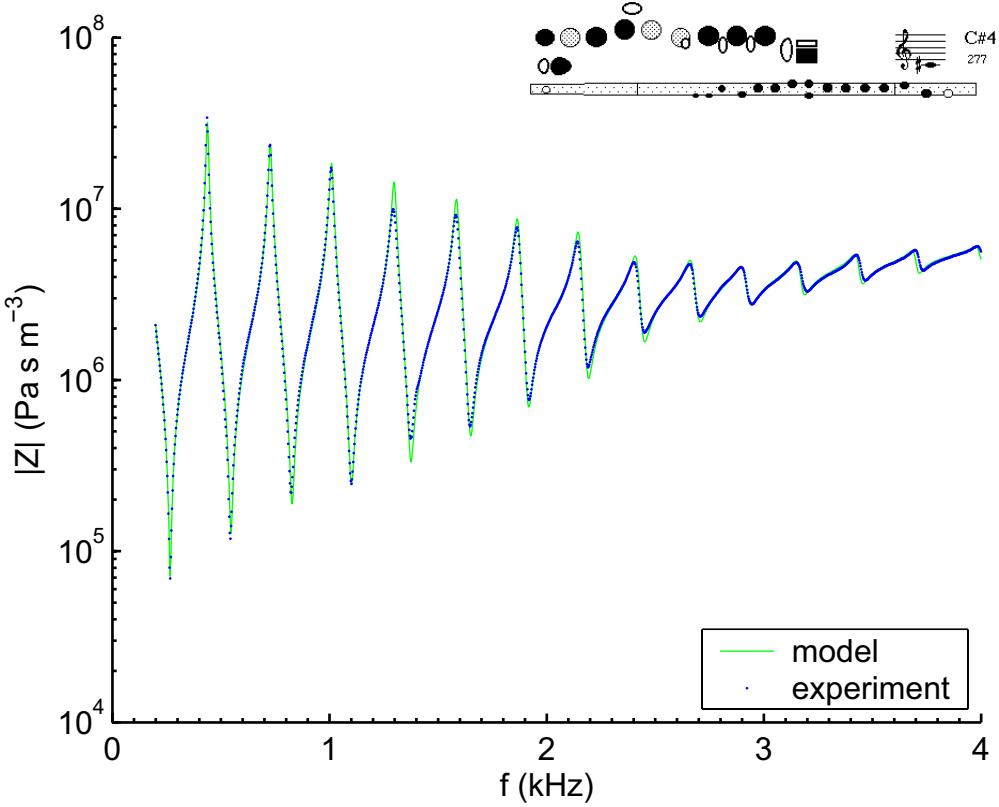


Figure 5.22: Impedance spectra (experiment and model) for the fingering C \sharp 4 on the modern flute using (2.30) to model the extra impedance caused by the overhanging keypad.

available for radiation and hence lead to a greater than expected length correction. For these reasons, an empirical correction may be required. Figure 5.22 shows the model fit for C \sharp 4 and Figure 5.23 shows the model fit for the fingering A4 (diffuse) with measured key dimensions (h , the height of the disk above the hole perimeter was taken to be that at the hole axis).

5.3.4.3 Two holes open

In the previous section, the model was used to model the impedance of fingerings with one open hole. The model was then tested on the fingering D4, in which two consecutive holes are open. Figure 5.24 shows the prediction of the model compared with experiment using the corrections applied in the previous section. The model overestimates the height and depth of impedance extrema, particularly in the region 1.5–2.5 kHz. For this reason, the resistance

$$R_{\text{open hole}} = 0.4 Z_0 (kb)^2 \quad (5.5)$$

was added to the radiation impedance of the open hole (k here is the wavenumber and b is the radius of the hole). Dalmont et al. (2001) do not address the effect of an overhanging keypad on the real part of the radiation impedance; the increase seen here (albeit indirectly) is physically quite plausible.

Inspection of Figure 5.25 reveals that over the frequency range 1.5–2.5 kHz the model locates the impedance extrema at frequencies slightly higher than measured. In this frequency range (encompassing the cut off frequency) the radiation impedance of open holes has the

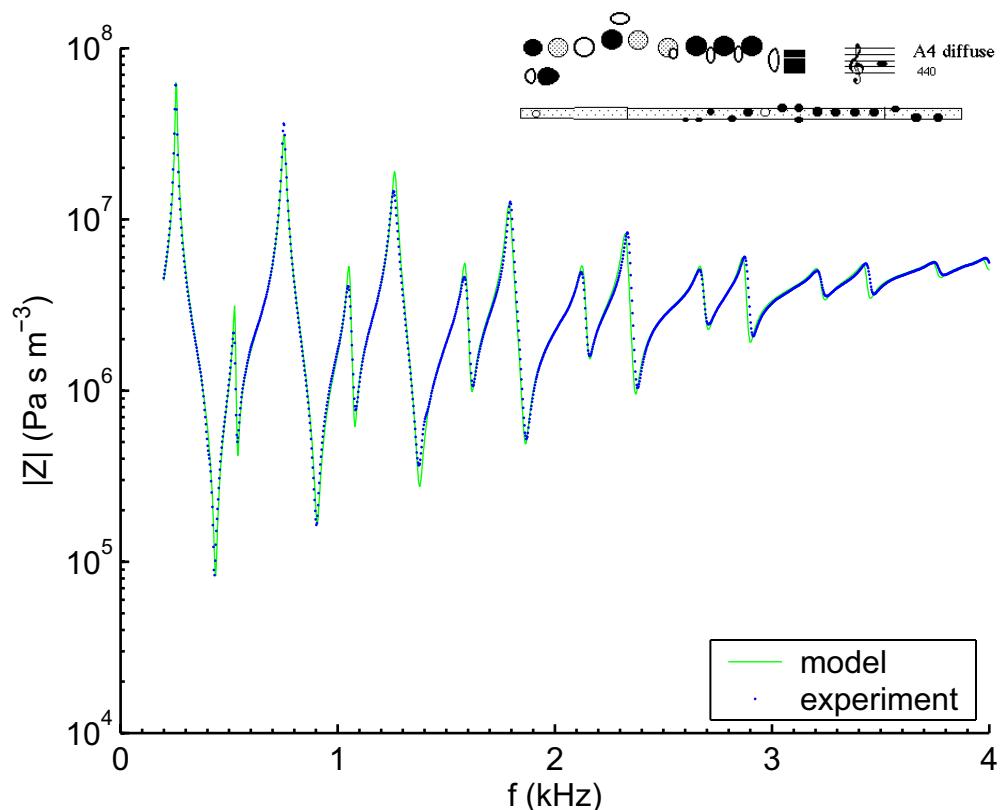


Figure 5.23: Impedance spectra (experiment and model) for the fingering A4 (diffuse) on the modern flute using (2.30 and 2.32) to model the extra impedance caused by the overhanging keypad.

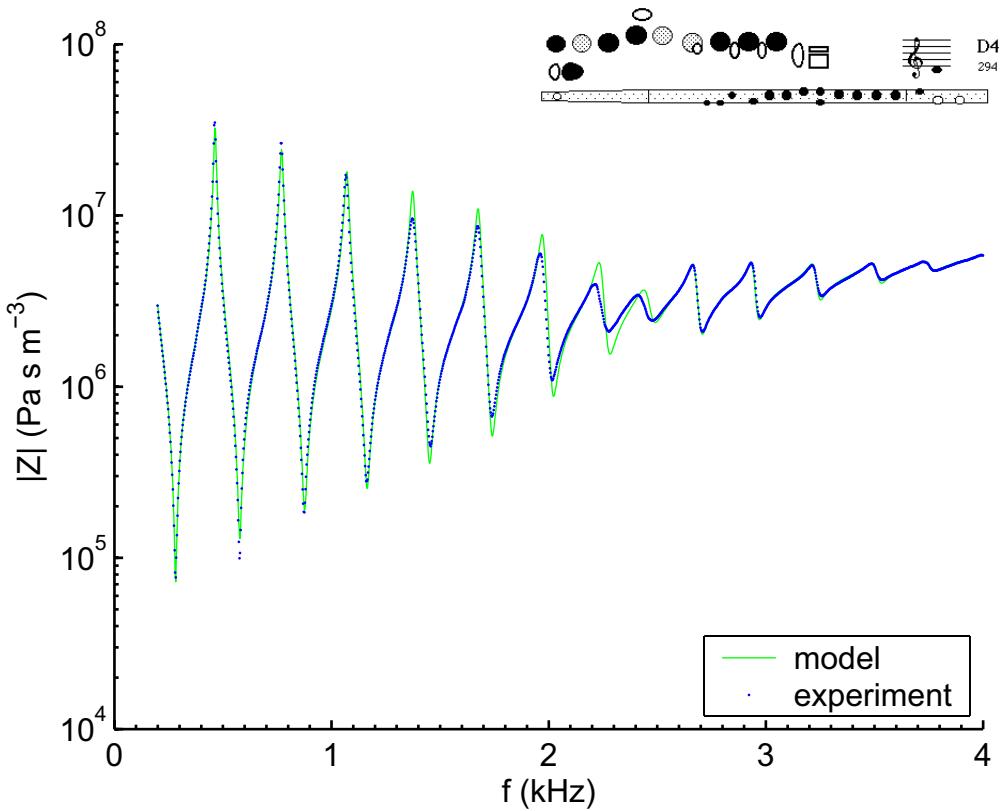


Figure 5.24: Impedance spectra (experiment and model) for the fingering D4 on the modern flute before correction for loss.

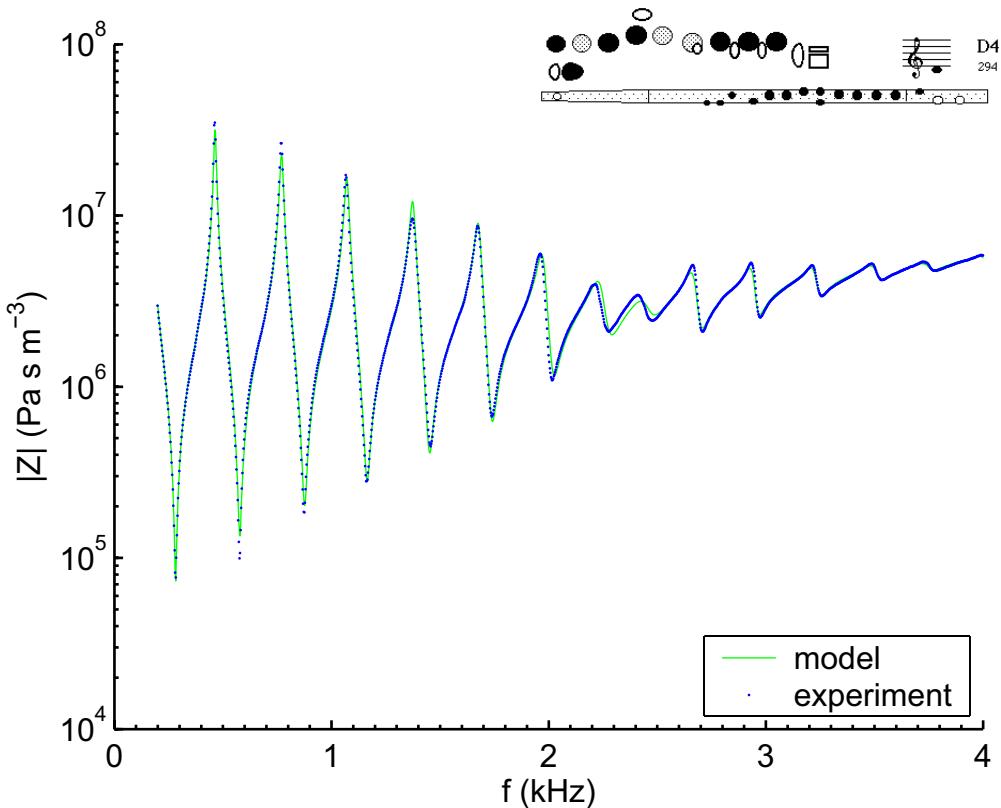


Figure 5.25: Impedance spectra (experiment and model) for the fingering D4 on the modern flute after correction for loss.

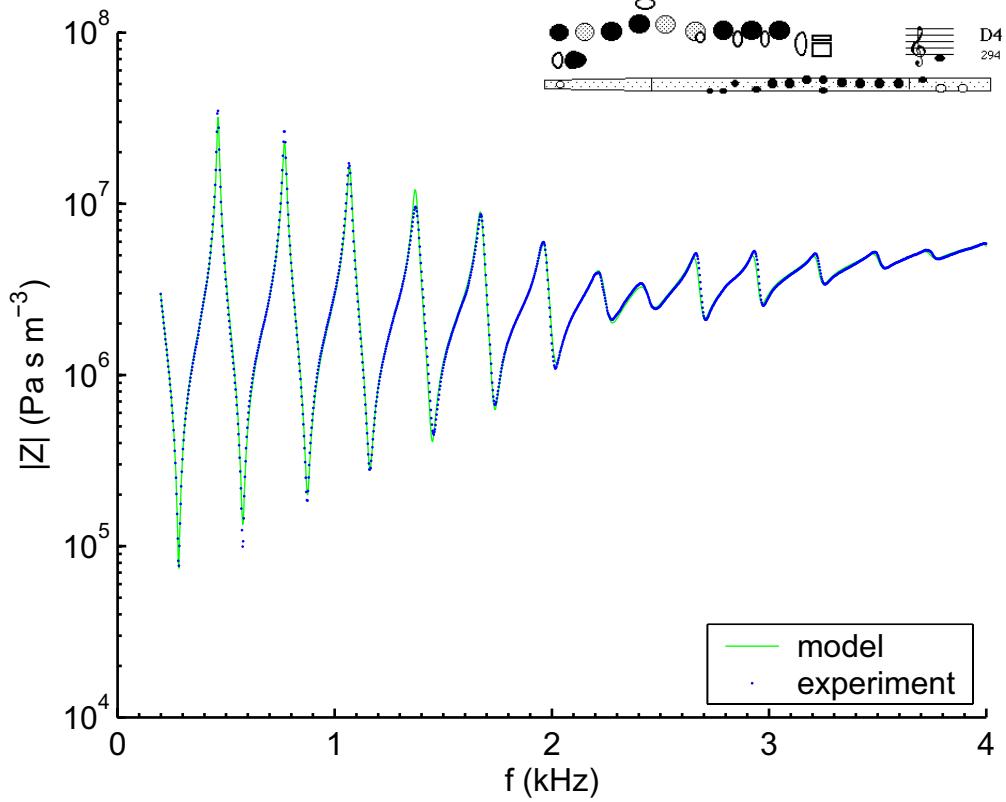


Figure 5.26: Impedance spectra (experiment and model) for the fingering D4 on the modern flute after correction for the extra radiation inertance.

greatest influence. In a real flute, the hole is ‘shaded’ not only by the key, but also by the body of the flute, and the keys themselves are not exactly like those modelled by Dalmont et al. (2001). For this reason the length correction

$$t_{\text{open hole}} = 0.1b, \quad (5.6)$$

was applied to all open holes and found to improve the model fit in this critical frequency range (Figure 5.26).

The optimised model was used to calculate the impedance for all standard fingerings; these are presented in Appendix A.

The model fits for C4, C♯4 and A4 (diffuse) are just as good or slightly better than those presented in this chapter (which show the performance of the model before all of the parameters were determined empirically).

5.3.5 Modelling the McGee classical flute

The McGee classical flute has eight finger holes and four keyed holes. Only six of the finger holes are used in fingerings; the two finger holes in the footjoint of the instrument cannot be reached by the fingers and are designed to improve the tone of the lowest notes. As discovered in Chapter 4, different length corrections need to be used for finger holes than for keyed holes, and the difference is more pronounced for closed finger holes. For this reason, we first test the accuracy of the model on fingerings without any closed finger holes or open keyed holes, and then proceed to include such fingerings.

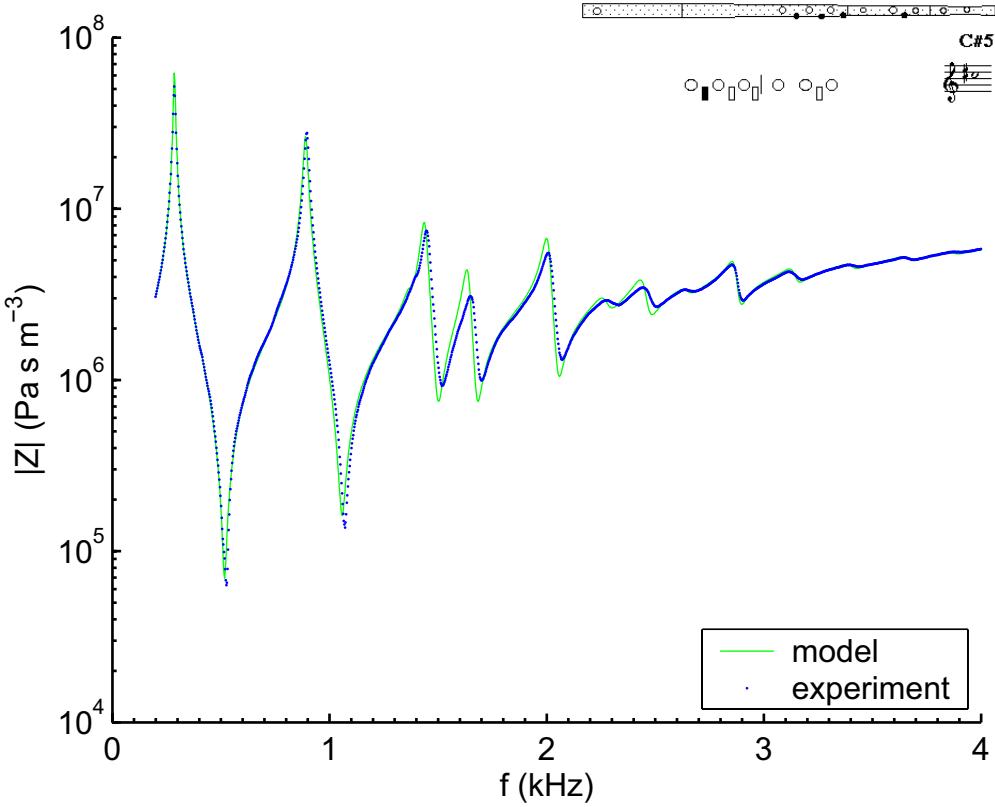


Figure 5.27: Impedance spectra (experiment and model) for the fingering C#5 on the classical flute without empirical correction.

5.3.5.1 All finger holes open

The fingering for C#5 on the classical flute has all finger holes open and all keyed holes closed. In Figure 5.27, the model is tested on this fingering using the equations in §2.2.10 and the low-frequency correction (2.29) to model the radiation impedance of open finger holes. The small length correction for closed keypads used in §5.3.4 is retained for classical flutes.

A better fit to the experimental data is obtained if we include a length correction for open holes of

$$t_{\text{open hole}} = -0.15b, \quad (5.7)$$

as shown in Figure 5.28. This length correction is opposite in sign to the length correction used for open keyed holes, which is not surprising—open finger holes do not have an overhanging keypad to increase the inertance. The small negative length correction may be in part due to the fact that finger holes are usually undercut to some degree.

5.3.5.2 Most finger holes closed

The fingering for D4 on the classical flute has all holes closed except for the two holes at the end of the instrument which are always open. In Figure 5.29 the model is tested on this fingering, but with the effects of closed holes deliberately ignored. As expected, the model prediction is clearly sharp compared to experiment. In Figure 5.30 the same fingering is modelled and the standard length corrections for closed (keyed) holes are used. This was done to test whether finger hole length corrections need be explicitly included in the model, or whether the

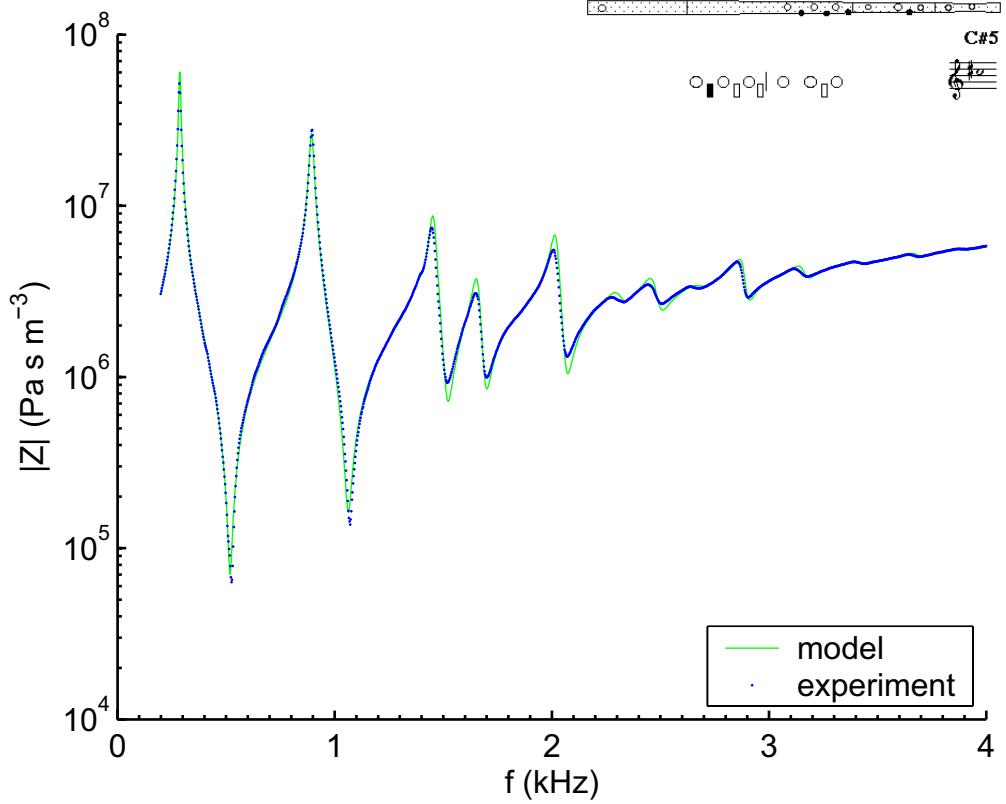


Figure 5.28: Impedance spectra (experiment and model) for the fingering C#5 on the classical flute with empirical correction.

correction used in §5.3.4 would suffice. As can be seen in Figure 5.30, this results in a model prediction that is too flat, particularly in the frequency range 1.5–2 kHz.

The length corrections identified in Chapter 4 for closed holes were then included in the model (Figure 5.31). The fit to the experimental data is better, justifying the use of explicit length corrections for closed finger holes. The overall flattening effect of closed holes is to some extent ameliorated for closed finger holes, since the protrusion of the finger into the hole reduces the hole shunt compliance, in some cases making it negative. The series impedance is also different for closed finger holes, although the effect of the series impedance is usually much less than that of the shunt impedance.

5.3.5.3 One keyed hole open

The fingering for F4 and F5 on the classical flute uses one open keyed hole. In Figure 5.32 the model is tested on this fingering, using the same length corrections for open keyed holes as were used for the modern flute. However, keyed holes on classical flutes usually do not have a chimney and so the radiation impedance of an infinite flange is more appropriate than that of a circular flange as used in §5.3.4. The model was altered to accommodate open keyed holes without chimneys. As can be seen in Figure 5.32, the model copes with the open keyed hole quite adequately.

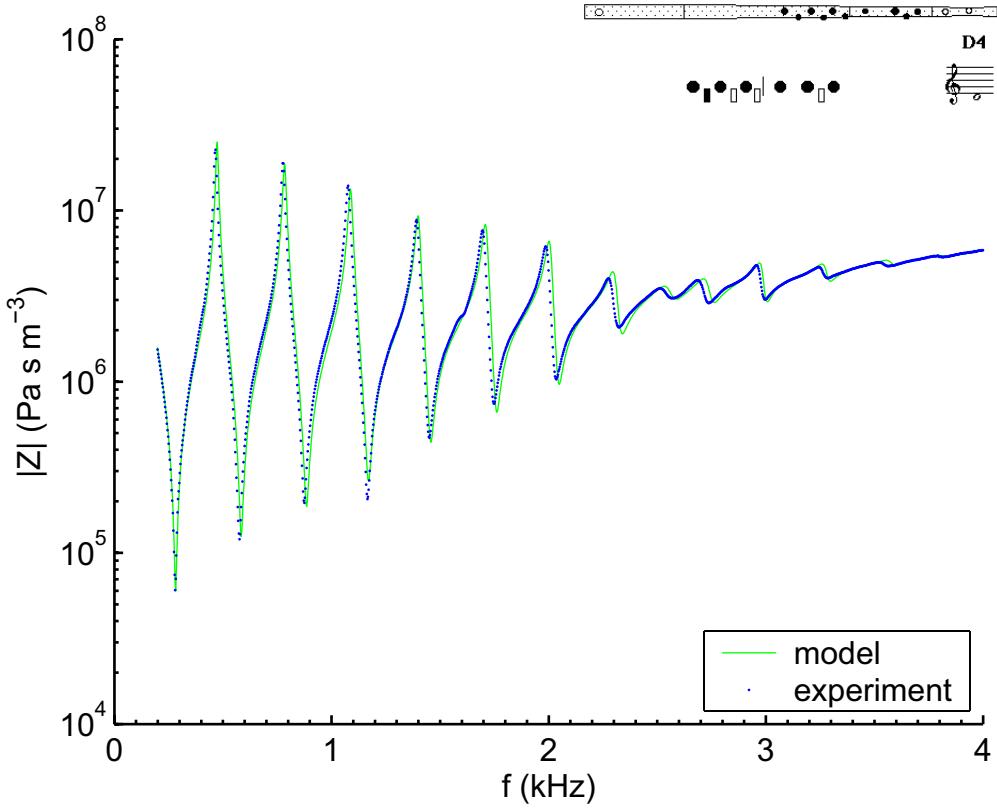


Figure 5.29: Impedance spectra (experiment and model) for the fingering D4 on the classical flute with no closed hole model.

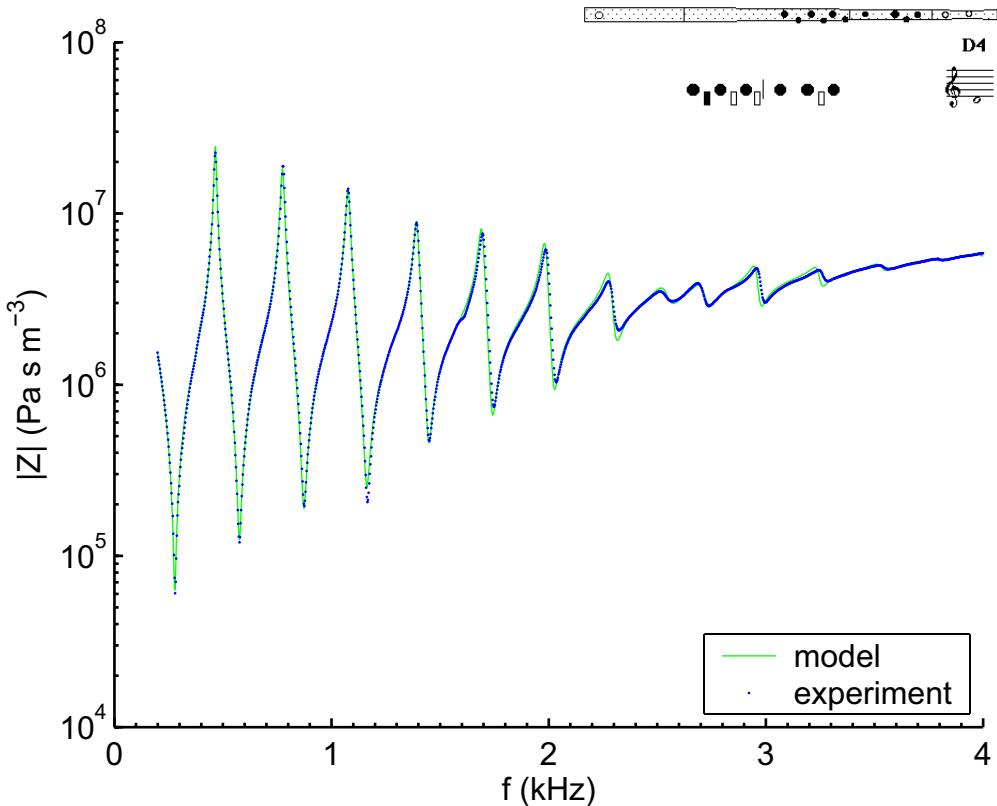


Figure 5.30: Impedance spectra (experiment and model) for the fingering D4 on the classical flute with the standard closed hole model.

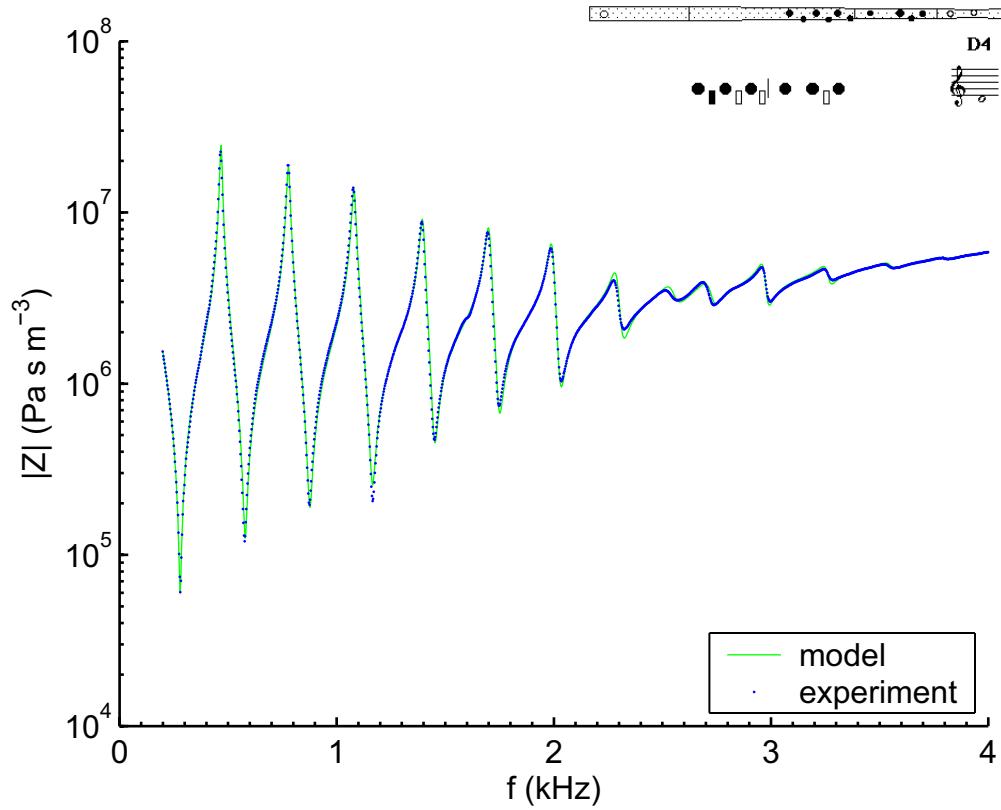


Figure 5.31: Impedance spectra (experiment and model) for the fingering D4 on the classical flute with an explicit model for closed finger holes.

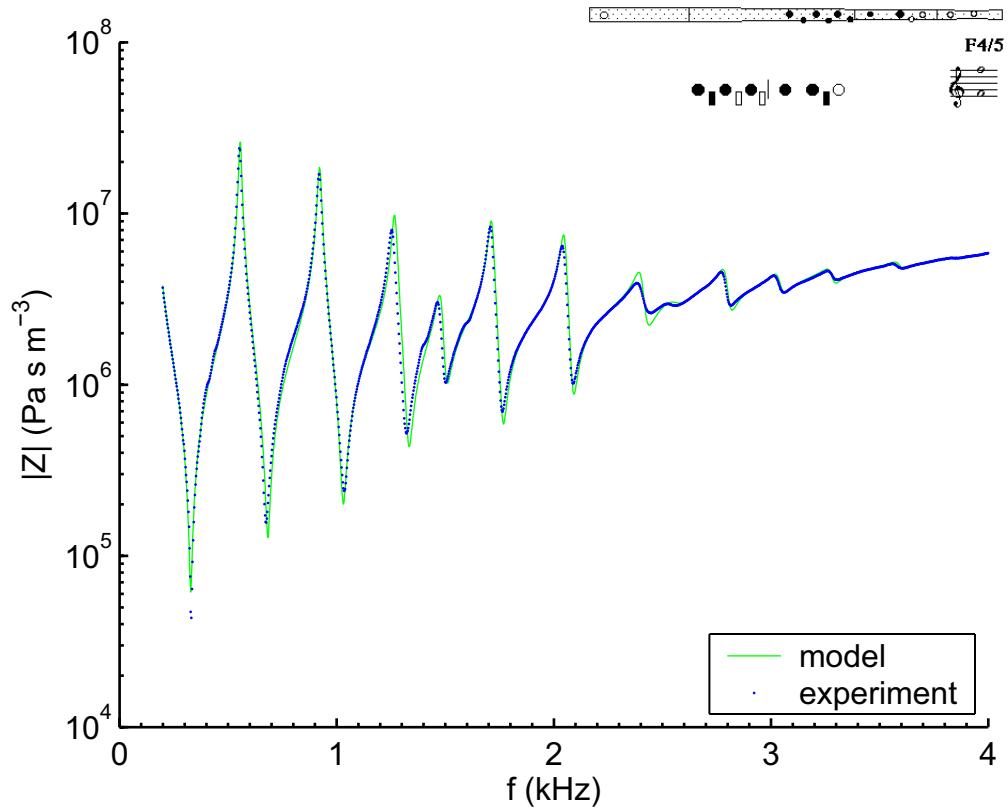


Figure 5.32: Impedance spectra (experiment and model) for the fingering F4/F5 on the classical flute.

5.3.5.4 *All recommended fingerings*

The model is compared with experiment for all recommended fingerings in Appendix A.

Chapter VI

Material and surface effects

6.1 INTRODUCTION

The material of which a flute is made is thought by some to affect the sound quality of the resulting instrument, but several studies (Coltman 1971, Widholm et al. 2001) have shown that players and hearers of the flute cannot distinguish between flutes made from a variety of different materials. Coltman compared keyless flutes made from silver, copper and grenadilla wood each played with the same headjoint made from Delrin plastic. Widholm et al. compared entire flutes of identical design made from solid silver, 9 carat gold, 14 carat gold, 24 carat gold and solid platinum, as well as platinum plated and silver plated flutes. For the study of Coltman, ‘no statistically significant correlation between the listeners’ scores and the material of the instrument body was found’ while Widholm et al. conclude that there is ‘no evidence that the wall material has any appreciable effect on the sound color or dynamic range of the instrument’.

The materials used in both of these studies were all good containers of air—i.e. they were non-porous or of very low porosity and they were able to be machined accurately and hold a good edge. In the study of Coltman the headjoint was the same for each flute body, so variations in the edge-holding ability of each material were not important. When entire flutes are made from wood, the situation is somewhat different. Different timbers will respond to machining in different ways and will have different surface characteristics and porosities. Different timbers are also likely to respond differently to playing and may change to a greater or lesser extent with time. The acoustic properties of timber flutes are reputed to change over time, particularly in the first few weeks of playing (see <<http://www.mcgee-flutes.com/newflute.html>>). These changes are most likely due to changes in the surface structure of the timber.

6.2 MATERIALS AND METHODS

In an attempt to evaluate the changes in a timber flute over the first few weeks of use, the impedance spectra of five wooden pipes were measured repeatedly over several weeks while the pipes were exposed to warm humid air.

The pipes were made from radiata pine, a plantation softwood timber. This timber is far too coarse, soft and porous for flute making. However it was used in this study to gain a qualitative appreciation of the effect of temperature, humidity and oiling on the impedance of timber flutes. The results are therefore an upper bound on the effects that should be expected for typical flute timbers. Each pipe was turned on a lathe to an outside diameter of approx. 20 mm and then drilled and reamed to a conical bore profile. The entry and exit diameters of the pipes were approx. 15 and 12 mm respectively and the pipes were approx. 120 mm in length. As part of this study, Terry McGee made a keyless classical flute from the same timber. The flute was played

Table 6.1: The treatment schedules for the five test pipes.

	Pipe 1	Pipe 2	Pipe 3	Pipe 4	Pipe 5
<i>Bore oiling</i>	none	none	none	once per week	once per week
<i>Warm humid air treatment</i>	none	23 hours per day	1 hour per day	23 hours per day	1 hour per day

before and after oiling as a qualitative test of the acoustics of such a non-optimal material.

The extent of the humid air exposure varied among the pipes, and some were oiled while others were not. The treatment schedules for the five pipes is summarised in Table 6.1. This schedule is designed to simulate an extreme range of playing conditions. Pipe 1 is the control. We would not expect to see much change in the impedance and surface characteristics of this pipe over the course of the experiment. Pipe 2 simulates the flute of a most negligent owner. The bore is not oiled and it is ‘played’ (i.e. exposed to warm humid air) constantly. Pipe 3 is ‘played’ for only 1 hour per day but never oiled. Pipe 4 is oiled diligently but ‘played’ constantly. Pipe 5 is oiled diligently and ‘played’ for only 1 hour per day.

In conformity to Terry McGee’s standard practice, the oiled pipes were oiled initially with raw linseed oil and thenceforth with a commercial bore oil. The oiled bores were allowed to dry overnight before humidity treatment was started or resumed. Once a day the humidity treated pipes were dried with a cloth swab. The air was humidified by bubbling it via a pump through water held at 32°C. Acoustic impedance was measured using the methods outlined in Chapter 3, using two microphones with a separation of 40 mm.

Once each week, the input diameter to each pipe was measured, the pipe was weighed, and the acoustic impedance spectrum of the pipe was measured.

6.3 RESULTS AND DISCUSSION

Figure 6.1 shows a single slice of a 3-dimensional image of a pipe, obtained using X-ray microtomography (micro-CT). The surface shown is perpendicular to the axis of the pipe and the slightly curved surface at the top of the image is part of the bore. Clearly visible are the wood cells and some deformation extending up to a dozen cells deep caused by the reamer that was used to cut the bore.

Terry McGee played the radiata pine keyless flute before and after oiling. Before oiling Terry could not play any notes below A4 (the flute was designed to play D4 with all holes closed) and the notes he could play were ‘weak and noisy’. In a test of the porosity of the timber, Terry closed all the finger holes and the end hole and was able to suck air through the walls of the instrument. Clearly, the attenuation of the acoustic waves with distance along the flute was much larger than that for a pipe with smooth walls. After oiling the flute all of the notes were playable, but they were weaker and more difficult to play than on a comparable instrument made from a more typical flute wood. Terry’s account of the experiment may be found on his web site at <<http://www.mcgee-flutes.com/FluteMyths.htm>>.

The pipes that were humidity treated for 23 hours each day increased in mass over the



Figure 6.1: A micro-CT image of a small bore section.

course of the experiment by a maximum of nearly 40%, with oiling making little difference. The pipes that were humidity treated for only 1 hour each day increased in mass by a maximum of 10% (for pipe 3) and by 5% (pipe 5). The input diameter of all of the humidity treated pipes decreased by approx. 3% as a result of the treatment, as the moisture raised the wood grain on the bore surface.

Selected acoustic impedance spectra for the five pipes are shown in Figures 6.2 to 6.6. In each figure the impedance measured before treatment is shown in black. The impedance measured after five weeks of humidity and oiling treatments is shown in red, and the impedance after one further week of drying is shown in green.

Exposing the bores to warm humid air (for prolonged or minimal duration) raises the grain significantly. The acoustic effect of this is seen in the impedance spectra of the two un-oiled but humidity-treated pipes. In Figure 6.4 (pipe 3) both impedances measured after treatment show a greater attenuation and the maxima and minima are shifted to the left in frequency. This is presumably due to the extra compliance of the rough surface. In Figure 6.3 (pipe 2) the attenuation decreases during treatment (red trace) and increases after the pipe has dried out. This shows that the roughness of the raised grain is ameliorated when the surface is wet since the water fills the bores of the wood.

The acoustic impedance spectra of the oiled pipes show the same general trend as those of the un-oiled pipes. Here however, the impedance spectra before and after treatment are roughly the same. This is despite visible raising of the grain caused by the humidity treatment, suggesting that the oil fills the pores of the wood and compensates for the extra attenuation caused by the raised grain. In Figure 6.5, we again see that the attenuation is significantly reduced when the wood is wet.

Samples were taken from stubs of each pipe at weekly intervals. These were imaged with moderate success using confocal microscopy, however the images obtained are difficult to interpret. I also investigated the use of micro-CT, a technique that uses X-rays to probe the sample (Figure 6.1), and scanning electron microscopy (SEM). For standard SEM the samples need to be dry—so it is impossible to investigate the surface characteristics of wet wood using this technique. Such a task could be accomplished using environmental scanning electron microscopy (ESEM), although this was not attempted for this thesis since the required equipment

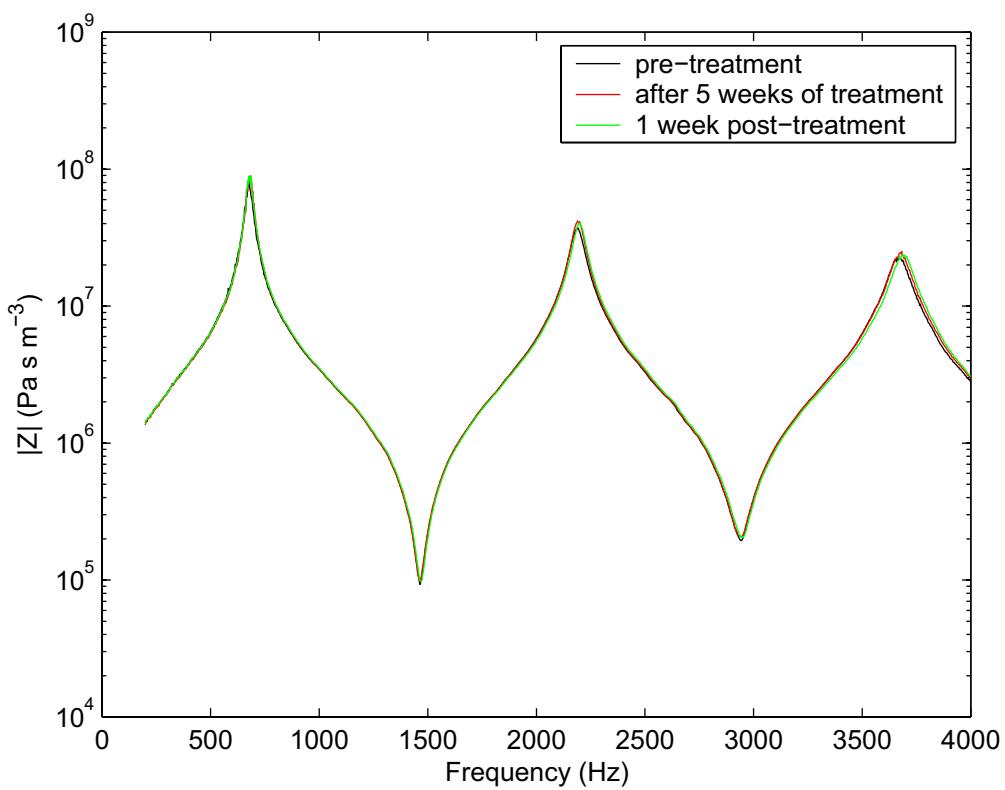


Figure 6.2: Impedance spectra for pipe 1 before, during and one week after treatment.

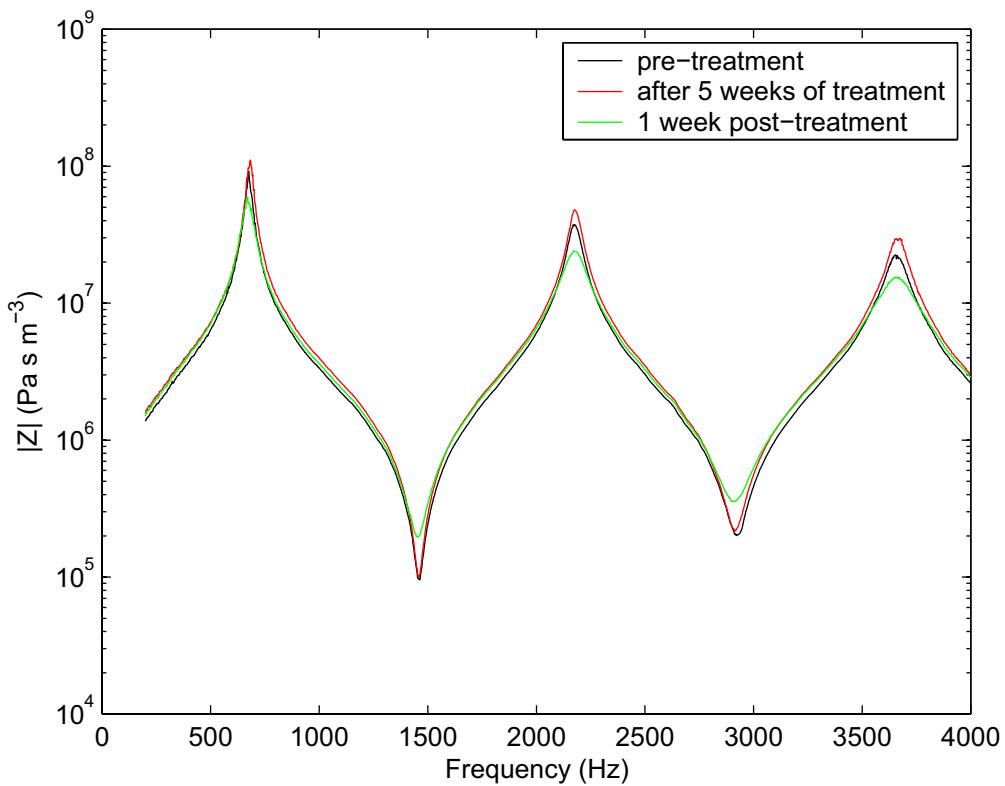


Figure 6.3: Impedance spectra for pipe 2 before, during and one week after treatment.

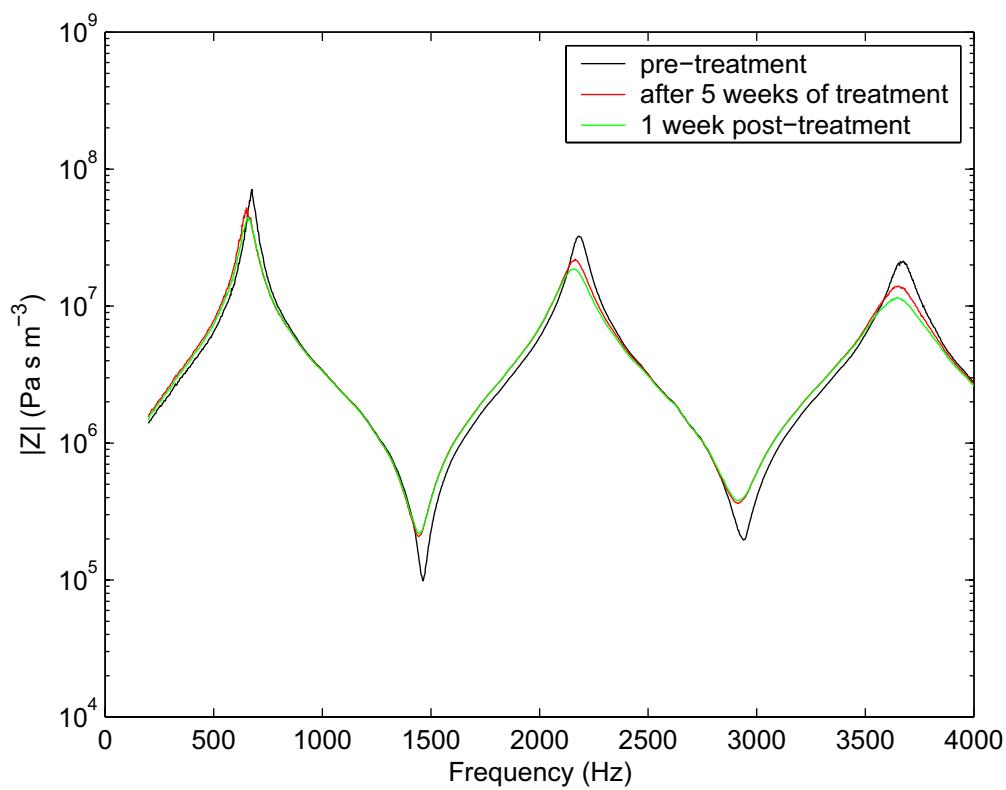


Figure 6.4: Impedance spectra for pipe 3 before, during and one week after treatment.

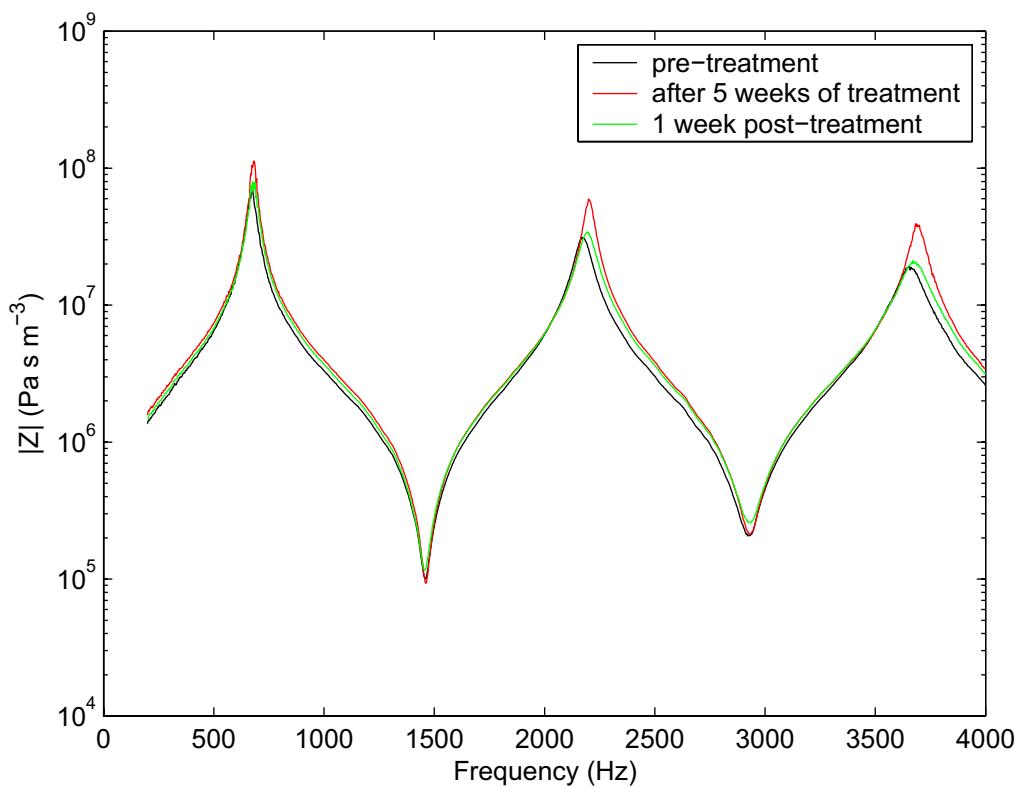


Figure 6.5: Impedance spectra for pipe 4 before, during and one week after treatment.

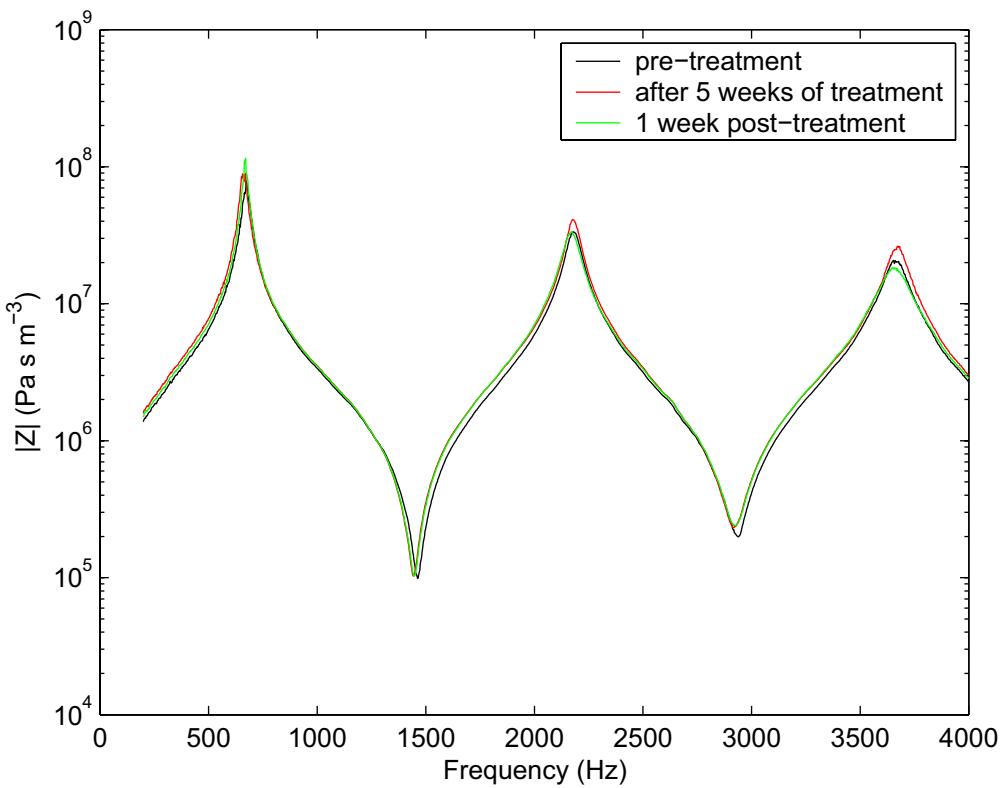


Figure 6.6: Impedance spectra for pipe 5 before, during and one week after treatment.

was not available at UNSW during the course of candidature.

6.4 FURTHER INVESTIGATIONS

In this study there seem to be three different contributors to the effect of surface conditions on acoustic impedance spectra. These are the macroscopic roughness (increased by humidity), how wet the surface is (increased by humidity) and whether the wood is oiled. These contributors are entangled in this study, and an attempt should be made to look at them in isolation.

In future work it may be useful to repeat this experiment with wood more suited for flute making. Any changes in acoustic impedance spectra would likely be of much smaller magnitude than found in this study, but would be more relevant to players and makers of wooden flutes. Such work may well answer questions about the best way to oil and play in a new instrument, especially if combined with surface microscopy (perhaps using ESEM).

Flute players and makers speak of gradual improvement in flute bores over time as they are repeatedly oiled and polished by the cleaning swab. In this experiment, none of the pipes showed less attenuation after treatment (with the possible exception of pipe 4). This is likely because of the huge macroscopic surface changes in a softwood like radiata pine when exposed to humid air. It may be possible to chart such changes in a hardwood such as is used in flute making.

Chapter VII

The embouchure hole and player corrections

So far, this thesis has been concerned only with the flute as a resonator—the measured impedance is passive and the interaction of the flute with the player has not been considered. Fortunately, and in contrast with the clarinet (Fritz & Wolfe 2005), the flute is largely decoupled from the player's vocal tract, and so it seems likely that changes in the latter do not affect much the resonances of the former. However, the interaction of the jet and player's face with the passive flute impedance needs to be considered, which is the subject of this chapter.

7.1 PLAYER IMPEDANCE CORRECTIONS

The flute impedance measured in Chapter 5 is the input impedance at the embouchure hole for the flute at room temperature as measured through a 7.8 mm diameter impedance head. However the impedance with which the jet interacts (and the impedance that should be modelled in order to derive meaningful tuning predictions) differs from the measured impedance in the following ways:

- the temperature of the air in the played flute is higher than room temperature and the temperature varies along the flute
- the humidity and CO₂ content of the air in the played flute are both higher than ambient air
- the player's lip covers part of the embouchure hole, changing the entry diameter. To first order, this is accounted for by the choice of impedance head. However, it varies among players and over the range
- the player's face shades the embouchure hole
- the acoustic wave can travel through the mouth into the vocal tract and thus the mouth impedance should be taken into account.

These effects can be measured or modelled (Thwaites & Fletcher 1983, Fabre & Hirschberg 2000, Vergez et al. 2005) but the physics is complex. For the current application it suffices to bundle these effects into a single impedance Z_{face} . In reality, these effects are not equivalent to a simple impedance, but this is an empirical model and such an approach is sufficient for the goals of this thesis. Fletcher (1976, see also Fletcher & Rossing (1998)) examines the jet-drive mechanism in organ pipes and shows that the radiation impedance from the mouth of an organ pipe (approximately equivalent to the partly-open embouchure hole of a flute) is in series with the 'internal' impedance. The jet itself contributes an active drive mechanism with both 'volume-flow' and 'pressure-drive' terms.

The impedance of the played flute is then

$$Z_{\text{played flute}} = Z_{\text{face}} + Z_{\text{flute}}, \quad (7.1)$$

where Z_{flute} is the input impedance of the flute with a temperature gradient and CO₂ and H₂O partial pressures in air equal to those in a played flute, as seen through an orifice with radius somewhat smaller than the input radius of the embouchure hole. The exact size of the orifice to use is difficult to determine, since the excitation of a played flute is not at all like the excitation used in Chapter 5, where the flute was excited with plane waves from the end of a 7.8 mm diameter impedance head. To get an idea of the extent to which the embouchure hole is covered by the player's lower lip, a flutist was asked to 'mime' several notes over the range of the flute. Miming should give representative results, since flutists are trained to regularly set their embouchure before playing, so that the initial transient of a note is strong and in tune. (The face impedance was measured simultaneously, but the results are erratic and do not show the expected variation with frequency.) Photographs of the embouchure for these notes are shown in Figure 7.1. It is clear from these photographs that the lower lip covers approximately 25% of the embouchure hole, and the upper lip protrudes somewhat further than the lower lip. The jet length appears to shorten slightly for higher notes. The jet length was estimated from the photographs to be 7.3 mm for C4 and 6.7 mm for F7—a decrease of 8%. For a more detailed study of the lip configuration of flute players see Fletcher (1975).

As was seen in §5.3.3, reducing the input radius of the embouchure hole lowers the frequency of impedance minima. Given the difficulty in determining the exact size of the input radius, a radius of 3.9 mm was used. This corresponds to the size of the impedance head used in Chapter 5, which in an earlier study was chosen to simulate the amount of the embouchure hole which remained uncovered by the player's lip (Wolfe et al. 2001a).

7.2 MODERN FLUTE TUNING

We choose to determine Z_{face} empirically based on tuning data. Three experienced flutists were asked to play each standard fingering on the same flute that was used for impedance measurements (Botros et al. 2006). The stopper and tuning slide of the flute were set at the same position as for measurements (i.e. 17.5 and 4 mm respectively). The flutists were asked to avoid correcting the pitch for notes that were out of tune; rather the goal was to achieve the best tone. As discussed in §2.2.12, the frequency of any note on the flute varies with the amplitude of excitation. For this reason the players were asked to play each note *mezzo forte*. The resulting pitch correction is then only strictly valid when the flute is played at this level; but a single pitch correction for each note is needed for flute design purposes, and the correction derived at *mezzo forte* is in my view most appropriate. The test notes were ordered so as to minimise the effect of pitch memory: the intervals between successive notes were large and mostly not simple consonances. Each note was repeated four times and the pitch was measured on a tuning meter with a resolution of 5 cents. The results are shown in Figure 7.2.

It is immediately obvious from Figure 7.2 that this modern flute in this configuration does not have perfect tuning, at least not for these players. There is a general trend towards sharper

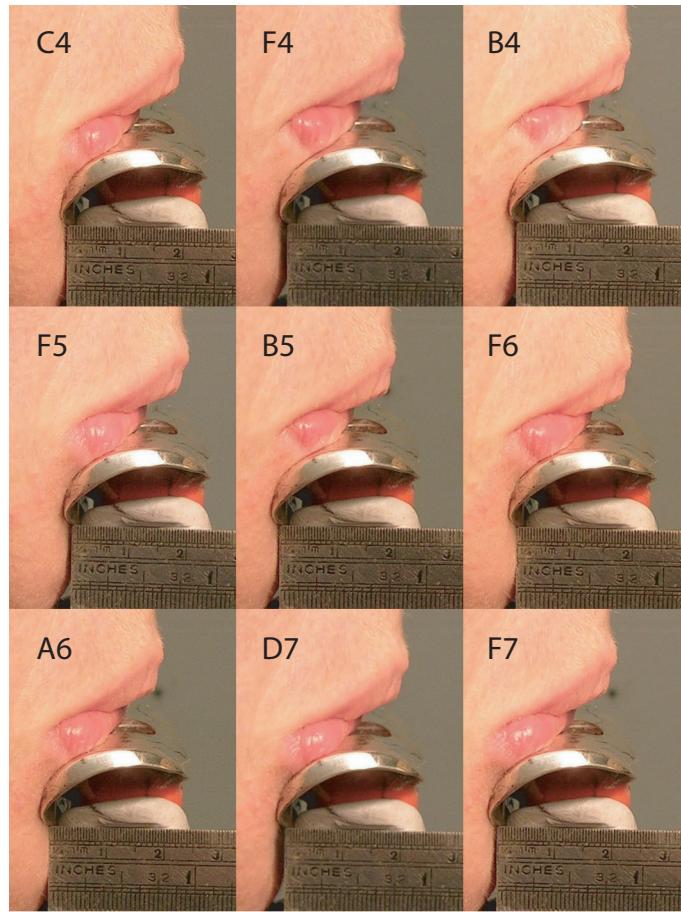


Figure 7.1: Photographs of the embouchure for mimed notes.

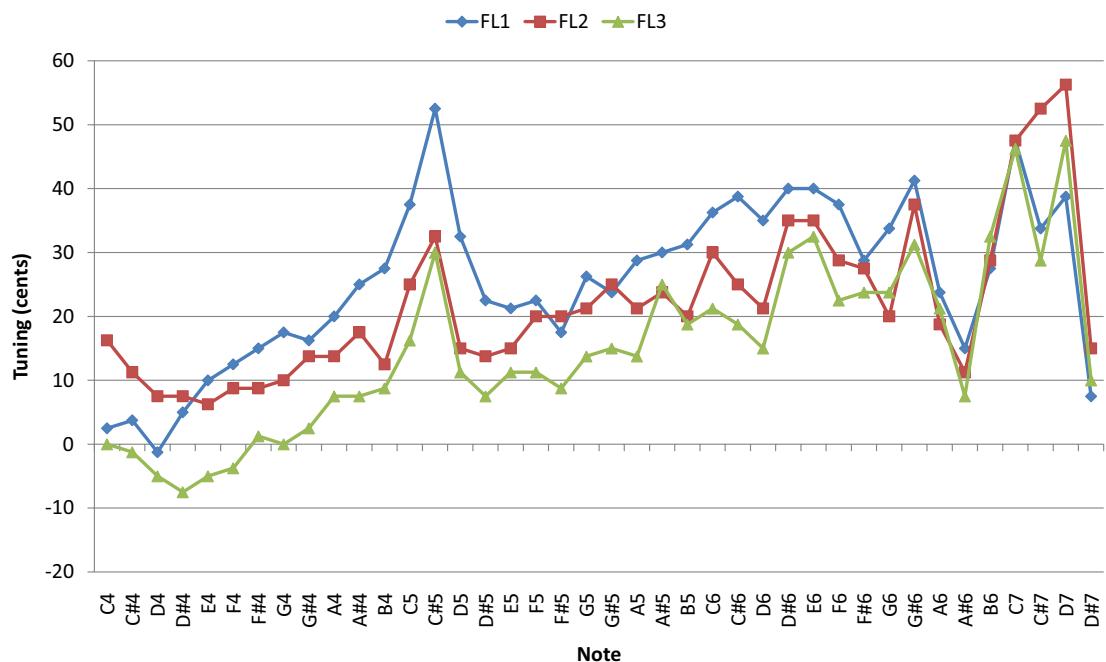


Figure 7.2: Tuning for three flutists over the range of the instrument. To reduce clutter, error bars have been omitted from this graph; the average range of tuning for four attempts at the same note was around 10 cents, with a maximum of 25 cents.

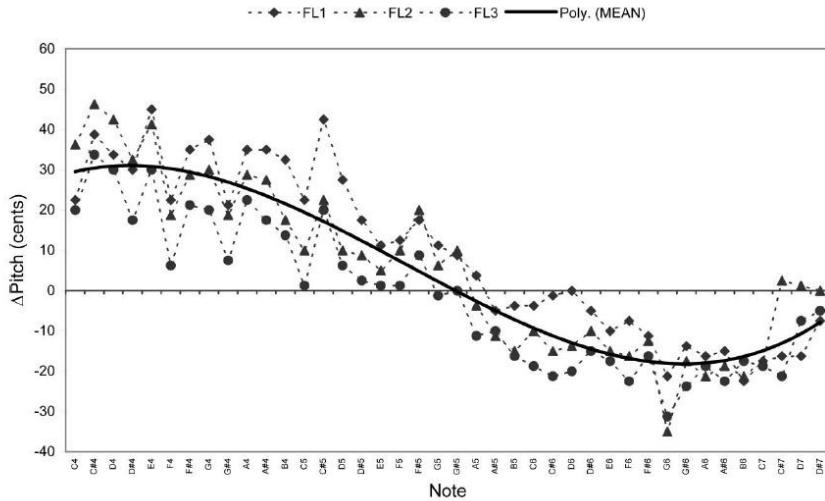


Figure 7.3: The pitch correction used by Botros et al. (2006).

tuning as the scale is ascended, and the tuning of some notes does not match the general trend (especially C \sharp 5 and A \sharp 6). The errant tuning of individual notes is easily explained: the placement and size of the holes on a flute is always a compromise and some tuning anomalies are tolerated in favour of keeping the flute design relatively simple and not too unwieldy. In particular the C \sharp hole doubles as tone hole for C \sharp 5 and C \sharp 6, and register hole for D5, D \sharp 5 and D6. This requires a compromise between position (higher than desired for C \sharp 5/6) and size (therefore smaller than desired for C \sharp 5/6). This compromise works better for C \sharp 6 than C \sharp 5 which is understandable—the pitch of C \sharp 5 is more easily adjusted by the player.

A player can always correct the tuning of individual notes with her embouchure. The trend towards sharper tuning over the scale is largely a function of the stopper position. The stopper position of 17.5 mm used here is a general guide—flute players adjust this parameter to correct the tuning—and there may well be a different optimum for the flute in question. Moreover, studies have shown that musicians actually like slightly wide octaves when not played simultaneously (Ward 1954) and this may explain the increase in tuning error with pitch of note. Incidentally, most of the notes in Figure 7.2 are sharp. This is due to the relatively short tuning slide position used in experiments. When informally playing the laboratory flute at A 440 our resident flutist, Jane Cavanagh, routinely set the slide closer to 8 mm than the 4 mm used here. Jane also has a fairly open blowhole (hers is the embouchure pictured in Figure 7.1).

For the Virtual Flute Botros et al. (2006) used a pitch correction to adjust the tuning derived from the impedance minima of measured spectra in order to match the data of Figure 7.2. The correction used by Botros et al. is shown in Figure 7.3.

7.3 CORRECTION FACTOR

In this thesis, Z_{face} is allowed to vary with played frequency in order to adjust the tuning. The input impedance of the flute is calculated with the temperature gradient given in §2.2.1 through

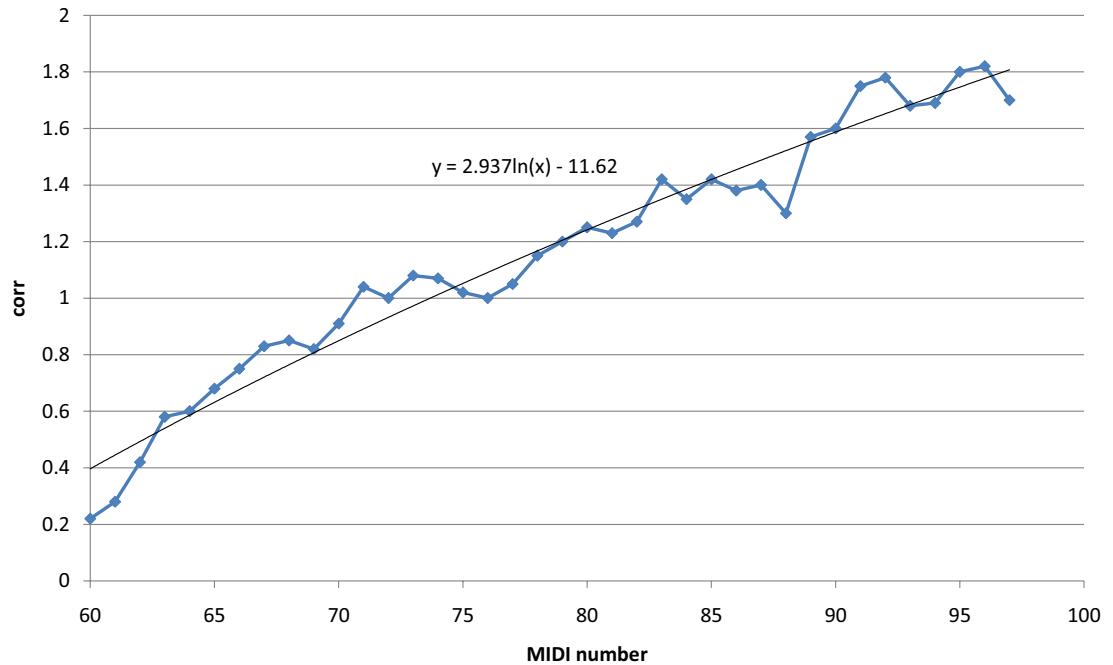


Figure 7.4: The correction factor $\text{Corr}(f)$ as a function of MIDI number (the plotted notes range from C4 to C \sharp 7).

an entry area equivalent to a circle of radius of 3.9 mm. Initially, Z_{face} is set equal to the impedance of a flanged pipe of radius 3.9 mm. This radiation impedance is then scaled by the factor $\text{Corr}(f)$, a correction factor fitted empirically so that the tuning predicted by the model matches (to the nearest cent) the average tuning of the three flutists. This correction factor is shown in Figure 7.4, with a logarithmic fit of the form

$$\text{Corr} = 2.9370 \log(m) - 11.6284, \quad (7.2)$$

where m is the MIDI number of the played note. Two outlying data points (for the notes D7 and D \sharp 7) were removed from the data set before the logarithmic fit was performed. This correction is valid for the three octaves from C4 to C \sharp 7—if one were to model flutes that play outside of this range (such as the alto flute or the piccolo) the tuning correction would need to be re-determined.

After applying the correction factor $\text{Corr}(f)$ to the radiation impedance of the embouchure hole, the predicted tuning of the modern flute differed from that measured by at most 10 cents (no worse than the tuning differences found between three flutists), but for most notes the accuracy was substantially better than this. The range of the correction factor is from 0.4 to 1.8, a much greater range than would be expected if the only mechanism involved is shading of the embouchure hole by the player's face.

7.4 APPLICATION TO THE CLASSICAL FLUTE

The considerations in this chapter have been about the player corrections for the modern flute. However, the FluteCAD software package is intended to predict the tunings for a wide variety

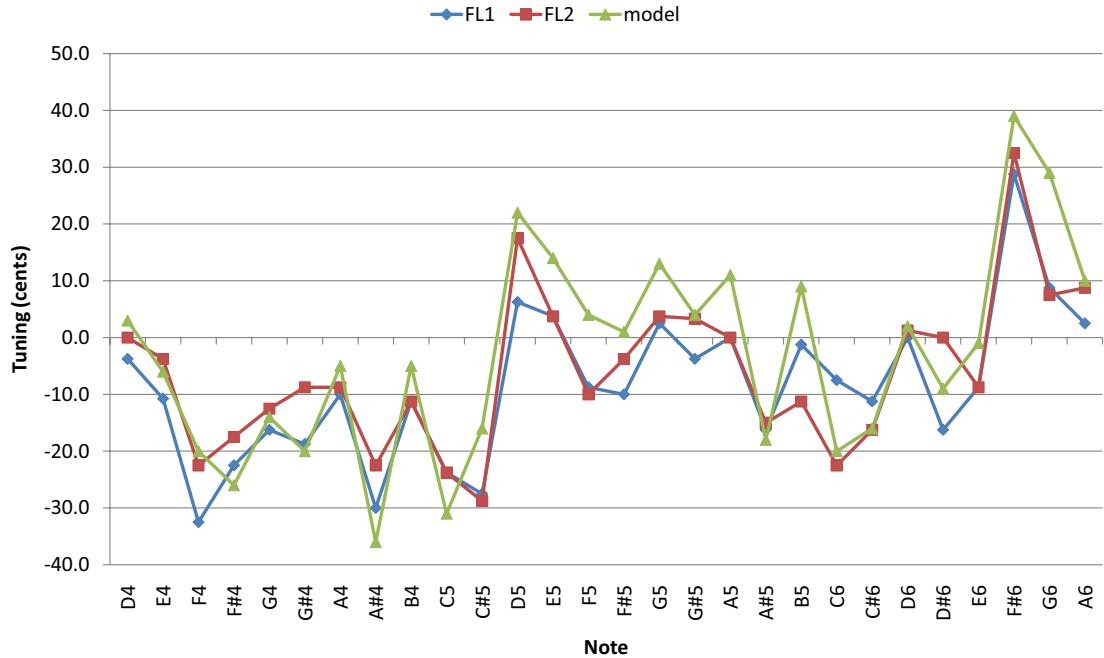


Figure 7.5: The tuning of the classical flute (experiment and model).

of flute geometries, including classical flute designs. Can the correction derived in this chapter be applied to the classical flute, or is another correction required? Since the sound production mechanism is the same in a wide range of different instruments, and the size and shape of the embouchure hole does not change a great deal, it is reasonable to expect that the correction derived here can be used universally. To test this assumption two flutists were asked to play the McGee classical flute without correcting for tuning. The pitches of the played notes were recorded and compared to the predictions of the corrected model. The results are shown in Figure 7.5. While the agreement between experiment and model is not exact, the general trends in the tuning has been well replicated. In the second octave the model predictions are rather sharper than measured, but by no more than 10 cents. The discrepancies between experiment and model in Figure 7.5 do not warrant a different model of the face impedance for classical flutes.

7.5 CONCLUSIONS AND FURTHER DIRECTIONS

In this chapter a correction factor was derived in order to empirically account for the effects on tuning of the player's face impedance and the jet impedance. The impedance spectrum was modified by the addition of an extra impedance element in the network representation of the instrument. Through the correction factor this extra impedance element changes over the range of the instrument. The final tuning predicted by the model is derived from the frequency of the impedance minimum nearest the frequency of the note. But since the jet excitation is nonlinear other impedance minima may contribute to the tuning of each note and the frequency of the best sound may be different at different levels of excitation. Further study is thus warranted on the relationship between tuning and acoustic impedance spectra. Further, the

player correction derived in this chapter is an average for a ‘typical’ player. However, there are significant differences between the playing styles of different flutists, and flutists playing in the classical or Irish traditions often use a different embouchure to players of the modern flute. It may be useful in the future to extend this study to take these differences into account.

Chapter VIII

Software implementation

This chapter describes the software implementation of the flute impedance model developed in the preceding chapters, and the development of a graphical user interface to the model.

8.1 THE IMPEDANCE MODEL

Command-line programs were developed using the software language C to calculate the impedance spectra of a flute, to analyse such spectra to derive musically relevant parameters, and to determine the pressure and flow profiles along the instrument. The geometry of the flute is described in an XML file. Much of this code derives from that of Botros (2001); his contribution is acknowledged in the code listings in Appendix B. The software implementation of the flute model is represented in the flow diagram in Figure 8.1. The various stages are described in turn.

8.1.1 Woodwind XML schema

All information about flute geometry is contained in a file written in Extensible Markup Language (XML). This language, of which HTML is a subset, contains data between descriptive tags. For example, a cylindrical bore segment of radius 9.5 mm and length 40.0 mm could be represented in XML as:

```
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>40.0</length>
</bore>
```

An XML file representing a woodwind (or a simple duct system) can be written manually using a text editor, or output from another computer program. To define a valid woodwind, an XML file must conform to certain rules. The rules are in the form of a Document Type Definition (DTD). The DTD for a woodwind (`Woodwind.dtd`) is given in Listing B.32. The elements `embouchurehole` and `upstream` defined in `Woodwind.dtd` are optional—for a woodwind other than a flute, or for a simple duct system, these elements are omitted in the XML description. XML files for the flutes and the clarinet discussed in this thesis are provided in Listings B.6, B.7 and B.15.

8.1.2 Woodwind data structure

The files `Woodwind.h` and `Woodwind.c` define a flexible structure that is used for calculating transfer matrices and impedances. The structure differs somewhat from that defined in `Woodwind.dtd`, mainly in order to improve performance.

For computational purposes, the flute is divided into several segments as suggested by

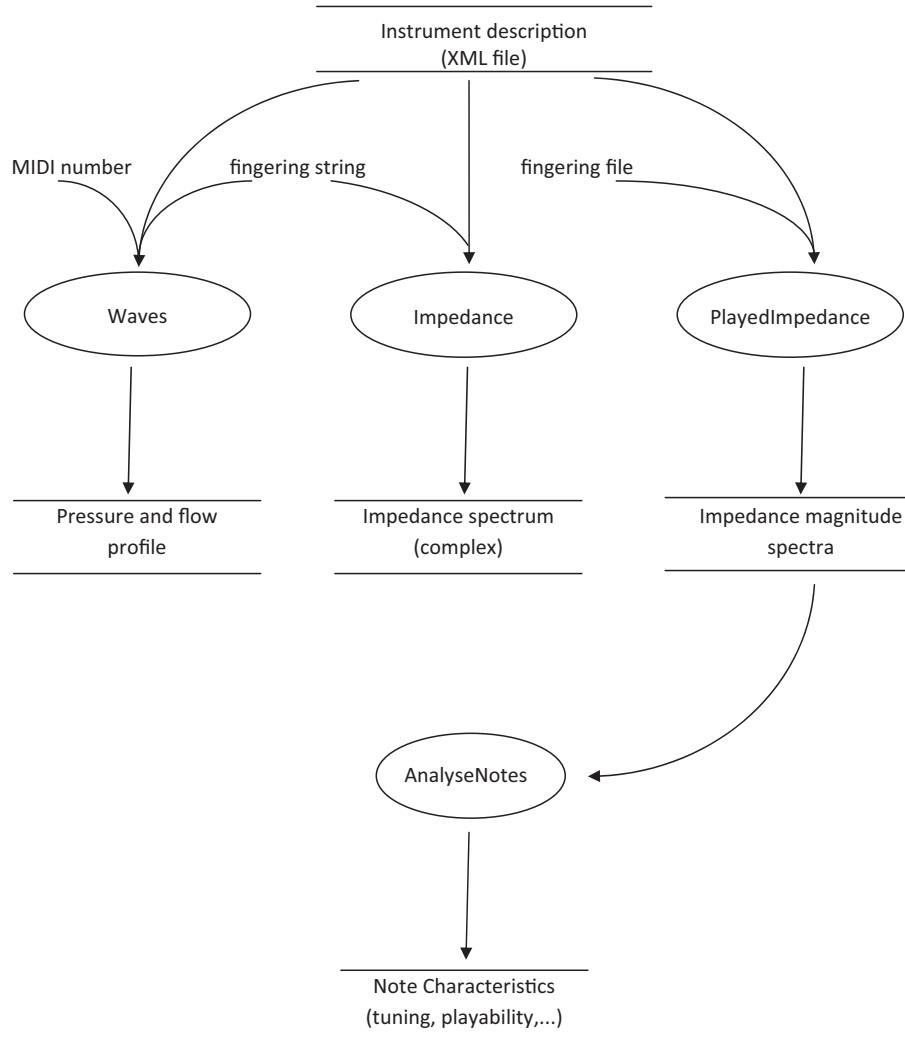


Figure 8.1: A flow diagram illustrating the programs described in this chapter.

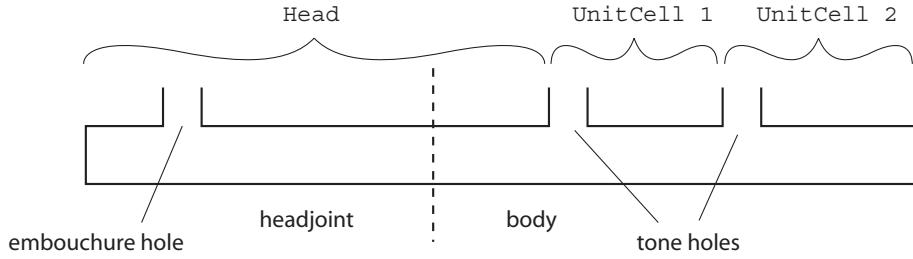


Figure 8.2: A schematic diagram of a ‘flute’ with two finger holes, showing the conceptual division of the instrument into a ‘head’ and two ‘unit cells’. Note that tone holes are modelled as lumped elements and that the various elements are divided at the midpoint of each tone hole.

Keefe (1990) (Figure 8.2). The first segment is the headjoint, which in this context includes the embouchure hole, upstream air column and downstream air column between the embouchure hole and first tone hole. The remaining segments are L-shaped unit cells which, as explained by Keefe, include a single hole and any bore section between the hole and the next hole downstream. This arrangement confers the distinct advantage that for each frequency the transfer matrices need only be calculated once. The impedance of the flute fingered differently may then be obtained by multiplication of saved matrices, thereby reducing computation time dramatically. A single transfer matrix relates the acoustic pressure and flow at the input of the instrument to those immediately upstream of the first tone hole. Two transfer matrices are calculated for each unit cell: one for the open hole and the other for the closed hole.

The Woodwind structure is defined in `Woodwind.h` as

```
typedef struct woodwind_str {
    Head head;
    Vector cells;
    double flange;
} *Woodwind;
```

The member `head` contains all of the flute dimensions upstream of the first finger hole, and `cells` is a linked list of ‘unit cells’, each describing one finger hole and any bore segments downstream before the next hole. The member `flange` is a dimensionless number related to the termination condition at the end of the instrument. It is equal to the wall thickness divided by bore radius for an open end. For a closed end (such as at the cork of a flute) `flange` is equal to -1.

The `Head` structure is defined as

```
typedef struct head_str {
    EmbouchureHole embouchureHole;
    Vector upstreamBore;
    double upstreamFlange;
    Vector downstreamBore;
    Map matrixMap;
} *Head;
```

and the `UnitCell` structure as

```
typedef struct unitcell_str {
    Hole hole;
    Vector bore;
    Map openMatrixMap;
    Map closedMatrixMap;
} *UnitCell;
```

The member `matrixMap` in `Head` and members `openMatrixMap` and `closedMatrixMap` in `UnitCell` are used by the program to store calculated transfer matrices for a particular frequency. The members `upstreamBore` and `downstreamBore` in `Head` and `bore` in `UnitCell` are linked lists of bore segments (cylinders or truncated cones). The member `upstreamFlange` in `Head` stores the termination condition for the upstream air column. For flutes the upstream air column is always stopped, so `upstreamFlange` is equal to -1. For the sake of generality `upstreamFlange` is provided as a parameter.

The structure `BoreSegment` is defined as

```
typedef struct boresegment_str {
    double radius1;
    double radius2;
    double length;
    double c;
    double rho;
} *BoreSegment;
```

where the members `c` and `rho` are provided to store the speed of sound and density of air for the segment. The remaining members are self-explanatory.

The embouchure hole is defined by the structure

```
typedef struct embouchurehole_str {
    double radiusin;
    double radiusout;
    double length;
    double boreRadius;
    double c;
    double rho;
} *EmbouchureHole;
```

where `radiusin` and `radiusout` are the inside and outside radii of the embouchure hole. Tone holes are defined by the structure

```
typedef struct hole_str {
    double radius;
    double length;
    double boreRadius;
    Key key;
    double c;
    double rho;
    char* fingering;
} *Hole;
```

The string `fingering` may be set either "OPEN" or "CLOSED". The member `boreRadius` in both the `EmbouchureHole` structure and the `Hole` structure is the radius of the bore at the position of the hole. The `Hole` structure may contain a `Key` member, which contains the relevant parameters describing the extent of shading of a hole by an open key:

```
typedef struct key_str {
    double radius;
    double holeRadius;
    double height;
    double thickness;
    double wallThickness;
    double chimneyHeight;
} *Key;
```

(For a tone hole without a key, `Key = NULL.`)

Functions implemented in `Woodwind.c` calculate transfer matrices of the various structures and the input impedance of the entire instrument. The `Woodwind` structure is used by the processes 'Impedance', 'PlayedImpedance' and 'Waves'.

8.1.3 'Impedance'

The process 'Impedance' calculates the input impedance of a woodwind defined by an XML file and is implemented in the C program `Impedance.c` (Listing B.10). The program has the following usage:

Usage: `Impedance [OPTIONS] <XML file>`

Options:

- s <holestring>
- t <temperature> (default 25.0 degC)
- u <humidity> (default 0.5)
- l <flo> (default 100.0)
- h <fhi> (default 4000.0)
- r <fres> (default 2.0)
- e <entryratio> (default 1.0)

<holestring>:

- Optional if no holes are defined in XML file.
- Must be a sequence of '0' (open hole) and 'X' (closed hole) characters.
- Length must be equal to the number of defined holes.
- e.g. "XX0000000X0000X000"

The mandatory argument `XML file` is the path to the file describing the instrument. This file is validated against the DTD file `Woodwind.dtd` which must be in the same directory as the XML file. Options are provided for setting the temperature (in °C), the relative humidity and the frequency range (Hz). The parameter `entryratio` allows calculation of the input impedance as seen through a radius smaller than the entry radius of the instrument. Thus `entryratio`

must be less than or equal to one. The parameter `holestring` sets the fingering for the instrument and consists of a sequence of '0' or 'X' characters (for open and closed holes respectively). For example, the following command would output an impedance spectrum for a modern flute:

```
Impedance -s XXXX00XX00XXXX0X -t 23.0 -h 0.8 ModernFlute.xml
```

'Impedance' parses the XML file into a Woodwind structure, calculates the speed of sound and density of air given the passed-in (or default) values of temperature and humidity, and calls the function `impedance(double f, Woodwind w)` in `Woodwind.c` for each frequency of interest. 'Impedance' then prints three columns of data: the frequency, and the real and imaginary parts of the impedance. Typical output looks like

2.000000e+002	1.103451e+005	2.442222e+006
2.020000e+002	1.149588e+005	2.495417e+006
2.040000e+002	1.198377e+005	2.550284e+006
...		

8.1.4 ‘PlayedImpedance’

The process 'PlayedImpedance' is similar to 'Impedance' but differs in several important ways. Firstly, the process calculates the input impedance of a *played* instrument; that is an instrument containing air of a composition more like exhaled breath than ambient air, and with a temperature gradient along its length (the properties of air in a played flute are discussed in §2.2.1). The temperature and humidity along the instrument are calculated by the model, and cannot be passed in as parameters. Secondly, the returned impedance is the sum of the instrument input impedance and the player (face) impedance (i.e. the radiation impedance as baffled by the player's face, especially lips); and the latter changes over the range of the instrument. Thirdly, the program is designed to calculate many impedance spectra at once, and the fingerings for the instrument are input as a file rather than as a string. Since the output of 'PlayedImpedance' depends on the note played (through the face impedance), and not just on the fingering, the input file is a list of tab-delimited MIDI numbers and fingerings, such as the following for a Boehm flute with C-foot:

60	XXXXXXXXXXXXXXXXXX
61	XXXXXXXXXXXXXXXXXO
62	XXXXXXXXXXXXXXXXOO
...	

The usage for 'PlayedImpedance' is as follows:

```
Usage: PlayedImpedance [OPTIONS] <input file> <XML file>
```

Options:

- l <flo> (default 200.0)
- h <fhi> (default 4000.0)
- r <fres> (default 2.0)

<input file>:

- Must be a tab-delimited list of midi numbers and holestrings, one set per line.

The output of ‘PlayedImpedance’ is a table of impedance magnitudes. The first column contains the frequency and the first row the MIDI numbers (aligned with the corresponding impedance vector). The impedance values are given in a logarithmic scale and are equal to $20 \log_{10} \frac{|Z|}{Z_0}$ where $Z_0 = 1 \text{ Pa s m}^{-3}$. The output for the input file snippet given above would look like

	60	61	62
200.00	124.178	126.977	129.918
202.00	123.710	126.512	129.414
204.00	123.241	126.053	128.925
206.00	122.770	125.599	128.448
208.00	122.296	125.148	127.981
210.00	121.818	124.701	127.524
...			

If the output from ‘PlayedImpedance’ is directed to a file, the file can then be used as the input to the process ‘AnalyseNotes’, which derives musically useful information for each note from its impedance spectrum. ‘PlayedImpedance’ is implemented in the C program file PlayedImpedance.c (Listing B.22) and most of the important functions are implemented in the files Woodwind.h and Woodwind.c.

8.1.5 ‘AnalyseNotes’

The process ‘AnalyseNotes’ is implemented in the C file AnalyseNotes.c (Listing B.3). ‘AnalyseNotes’ calls functions in Minima.h and Minima.c (Listings B.13 and B.14) to retrieve the frequency, impedance and bandwidth of impedance spectrum minima and calls functions in Analysis.h and Analysis.c (Listings B.4 and B.5) to determine (among other things) the playability and strength of the impedance minima. These programs are largely the work of Botros (2001), and detailed documentation is available in his thesis. The interfaces for the programs have been modified slightly for this thesis and minor changes have been made to some files.

‘AnalyseNotes’ searches for minima in each impedance spectrum in the passed-in file, and determines the frequency, impedance and tuning of each minimum. These quantities are approximated from a parabola fitted to eleven data points centred on the minimum and spanning a frequency range of 20 Hz. If the impedance curve corresponding to some played note has a playable minimum nearest to that note, then the minimum is analysed and the playability, strength and harmonicity are output along with the frequency and tuning. The playability, strength and harmonicity are calculated using an expert system developed by Botros et al. (2002, 2006) and trained on the Acoustic Laboratory’s resident ‘expert’, flutist Jane Cavanagh. The usage of ‘AnalyseNotes’ is

```
Usage: AnalyseNotes [OPTIONS] <filename>
```

Options:

-h (Displays harmonicity data)

and the output line for a single note contains the information

```
<midi> <playability> <strength> <frequency> <tuning> <Z>
      [numHarm] [meanHarmZ]
```

Typical output is

60	3.0	3.9	261.8	1	98.4	12	104.0
61	3.0	3.9	276.8	-2	98.3	12	104.3
62	3.0	3.8	293.4	-2	98.2	11	104.5
...							

A playability value of 3.0 indicates that the note is easily playable, while 1.0 is difficult to play. Likewise, a note with a strength of 4.0 is strong and bright while 1.0 is weak and dark.

8.1.6 ‘Waves’

The process ‘Waves’ calculates and outputs the relative magnitudes of pressure and flow along the bore of an instrument. The usage is as follows:

```
Usage: Waves [OPTIONS] <midi> <frequency> <XML file>
```

Options:

- s <holestring>
- r <xres> (default 2.0)

<holestring>:

- Optional if no holes are defined in XML file.
- Must be a sequence of '0' (open hole) and 'X' (closed hole) characters.
- Length must be equal to the number of defined holes.
- e.g. "XX000000X0000X000"

The required argument `midi` is the MIDI number for the played note and `frequency` is the frequency of interest (usually a resonant frequency, as obtained from ‘AnalyseNotes’). ‘Waves’ is implemented in the C program file `Waves.c` (Listing B.29) and relies on functions in `Woodwind.h` and `Woodwind.c`.

‘Waves’ calculates the pressure p_{in} and volume flow U_{in} at the input to the played instrument in relative units such that $|p_{in}|^2 + |Z_0 U_{in}|^2 = 1 \text{ Pa}^2$, where Z_0 is the characteristic impedance at the input to the instrument. The pressure $p(x)$ and volume flow $U(x)$ are calculated at points along the instrument bore `xres` mm apart. The output of the program is a list of tab-delimited x , $|p(x)|$ and $Z_0|U(x)|$ triplets, such as

-16.0	0.111	0.002
-14.0	0.111	0.004
-12.0	0.111	0.006
-10.0	0.111	0.009
-8.0	0.112	0.011
-6.0	0.112	0.013
-4.0	0.112	0.016
-2.0	0.112	0.018
0.0	0.110	1.619

Table 8.1: C library files for the woodwind model. C program files (.h and .c) for each filename are provided in Appendix B.

Filename	Description
Acoustics	Calculates acoustical constants, transfer matrices etc.
Analysis	Contains methods used by AnalyseNotes.
Complex	Complex arithmetic library.
Map	Simple implementation of a map (associative array).
Minima	Contains definition and methods for finding minima.
Note	Representation of a musical note, including name, tuning and midi number.
ParseImpedance	Parses the file returned by PlayedImpedance for use by AnalyseNotes.
ParseXML	Parses XML representation of a woodwind, and returns a Woodwind structure.
Point	Definition of a simple structure containing (x, y) data.
TransferMatrix	An implementation of a 2×2 matrix, and functions to multiply and invert such matrices.
Vector	A linked list of void* pointers.
Woodwind	Definition of Woodwind structure, and functions to calculate transfer matrices and impedances.

2.0	0.117	1.620
4.0	0.124	1.620
...		

(The origin of the x-axis used by ‘Waves’ is at the embouchure hole of the flute; hence the negative values.)

8.1.7 Library files

Various library files are used by the programs described in this section. A brief description of each of these is given in Table 8.1 and the complete program listings are given in Appendix B.

8.2 THE USER INTERFACE

The previous section described the development of a set of command-line tools to calculate the acoustical and musical properties of a flute. These tools incorporate the mathematical model of the flute. In order to make the results of this thesis generally accessible to members of the flute making community, who may not have a scientific background, a user-friendly interface to the command line tools was developed, using the computer programming language Java. Java is a language developed by Sun Microsystems with similar syntax to C and C++, but Java applications are typically compiled to run on *virtual machines*. Programs developed using Java are easily deployed to different machines with different operating systems, since Java Virtual Machines (JVMs) are available for most popular platforms.

8.2.1 Design goals

The following attributes were considered highly desirable in the user interface:

- graphical display of the modelled instrument
- dimensional representation of the instrument familiar to a maker and not too unlike conventional descriptions
- instrument files that could be saved (to form part of a maker's compendium or 'toolbox' of designs)
- graphical display of tuning data (most important) and playability and timbral data (less important but still desirable)
- display of pressure and flow profiles along the instrument, to aid placement of e.g. register holes and to inform the maker's intuition
- simple, intuitive layout.

The remainder of this chapter describes the layout and design of the user interface. The Java code has not been included in this thesis since the software may be made commercially available and the owners of the intellectual property do not wish the complete source code to be published at this stage. The reader is invited to contact the author for up-to-date information.

8.2.2 Brief description of application features

A screen shot of a window in the user interface 'FluteCAD' is shown in Figure 8.3. The 'Design' tab of the window is active and the body joint of a modern flute is displayed. The window contains a schematic diagram of the instrument (top) and a panel of dimensions (bottom left). A user navigates between the different instrument components using labelled tabs. Length dimensions are in millimetres and measured from either the centre of the embouchure hole (for the head joint) or from the upstream end of the bore (for body joints). Holes are described by their diameter, length and angular position on the flute body (relative to that of the embouchure hole). If a hole has a key, the dimensions of the key are likewise entered. The bore of each instrument joint is described by two or more points joined by straight lines. The application uses a detailed set of validation rules to ensure that any changes made to the flute design do not result in an invalid instrument. (An invalid instrument may have e.g. holes too big to fit within the bore.) Any invalid requests are rejected by the application and a message is communicated to the user via an error message bar at the bottom of the window.

The 'Fingering Chart' tab of 'FluteCAD' is shown in Figure 8.4. For each note the fingering is an array of booleans, with a closed hole represented by a `true` and an open hole by a `false` literal. When the 'Tuning' tab of the user interface is selected, the application calculates the impedance spectra for all fingerings (using 'PlayedImpedance') and analyses these spectra for playable notes (using 'AnalyseNotes'). The tuning results are displayed both graphically and in tabular form (Figure 8.5). Each octave is displayed as a separate series, so that notes with the same or similar fingerings (e.g. G4 and G5) can be easily compared. Analogous graphs of the playability and strength of each note are given on separate tabs.

The final tab gives detailed information about a single (selectable) note. In Figure 8.6 the note display tab for the note D7 is shown. The pressure and flow profile for the note is given,

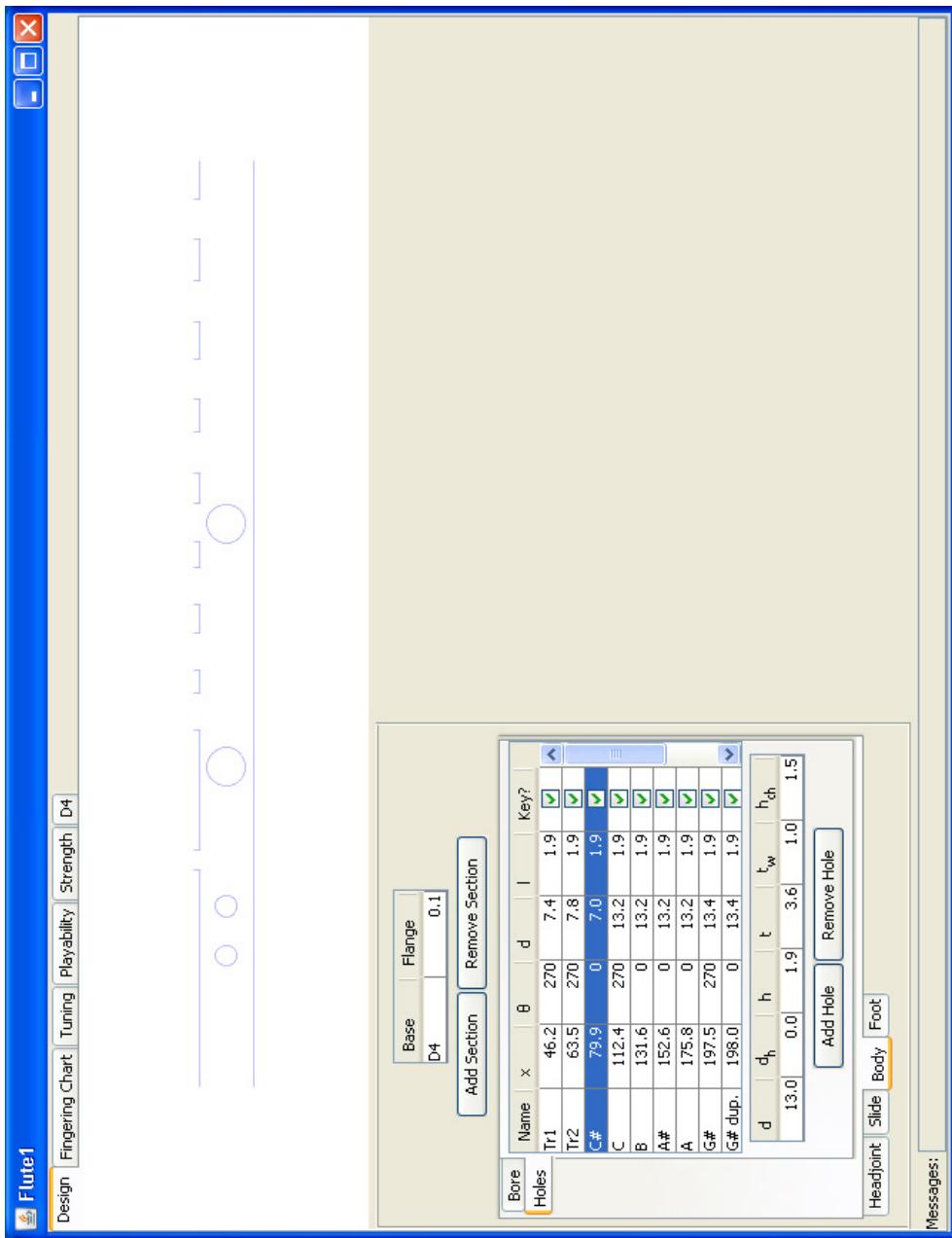


Figure 8.3: A screen shot of the 'Design' tab of the user interface.

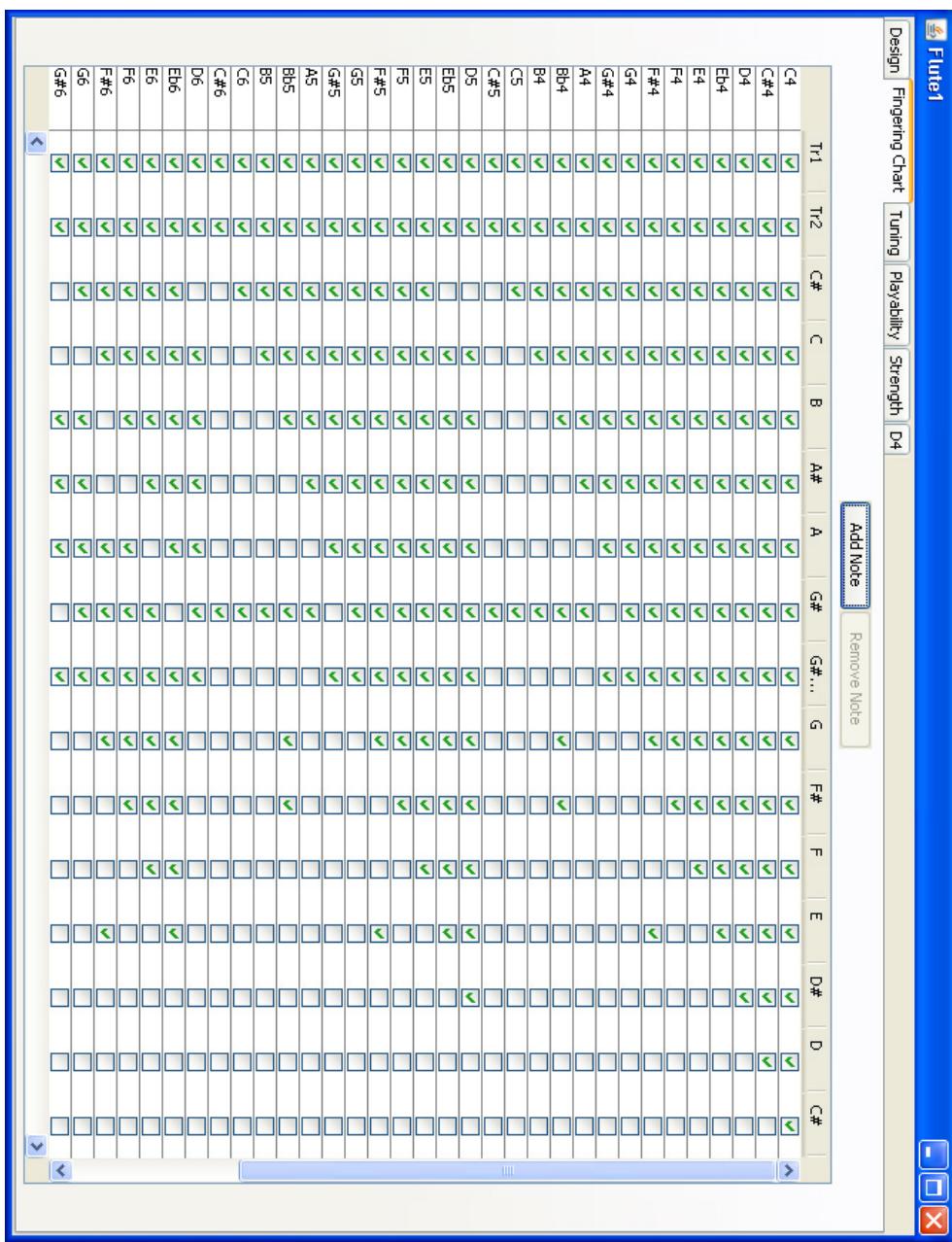


Figure 8.4: A screen shot of the 'Fingering Chart' tab of the user interface.

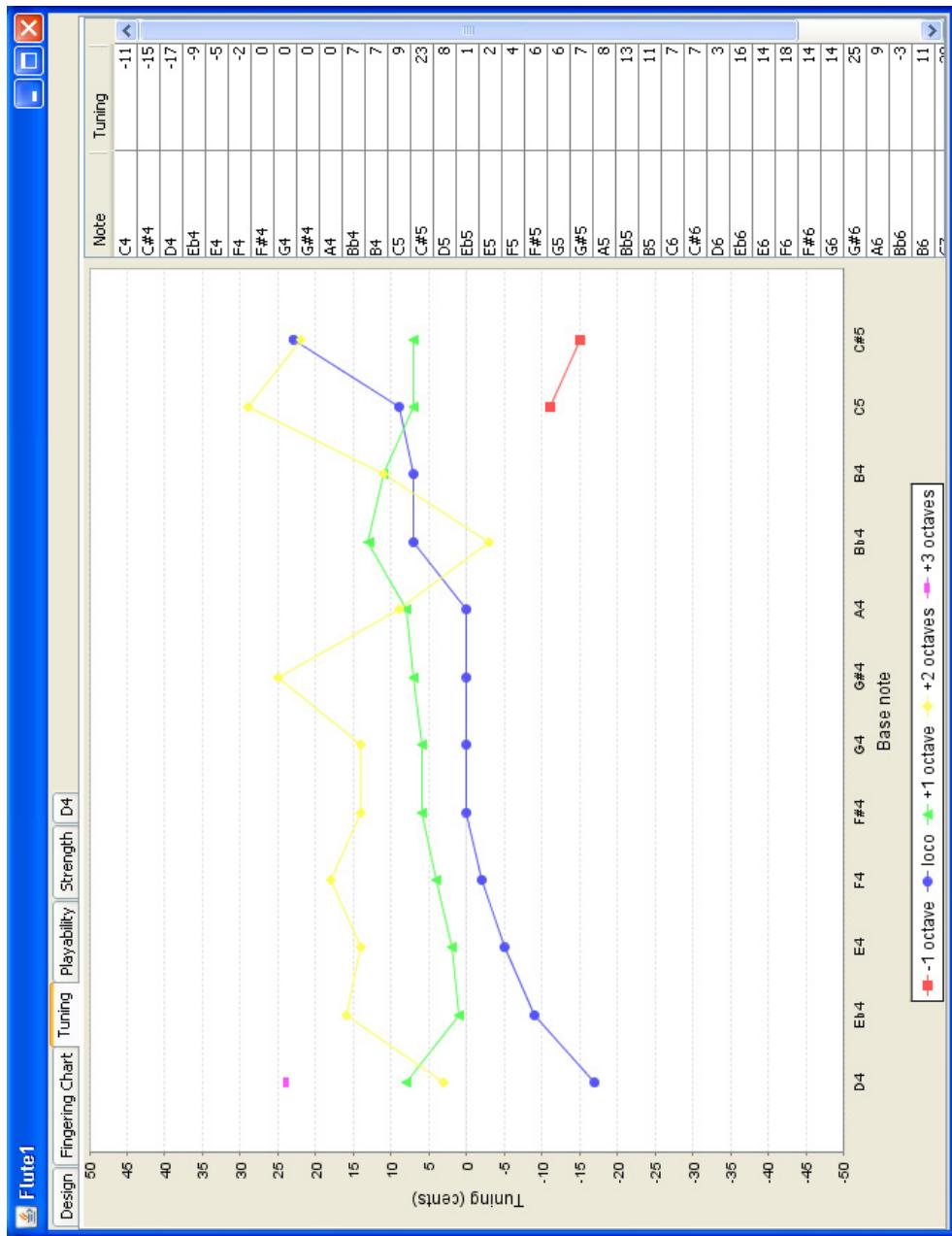


Figure 8.5: A screen shot of the ‘Tuning’ tab of the user interface. The tuning displayed is for the modern flute with C foot and tuning slide at 8 mm.

along with the impedance spectrum and various data relating to the analysed minimum (shown with a black arrow on the impedance spectrum). For the note D7, the acoustic waves have significant energy content over the entire length of the flute, despite several open holes. In this range of the instrument most fingerings are cross-fingerings, since no one hole controls the frequency of the played note. The calculated pressure and flow profiles are very useful for flute makers in tuning cross-fingered notes.

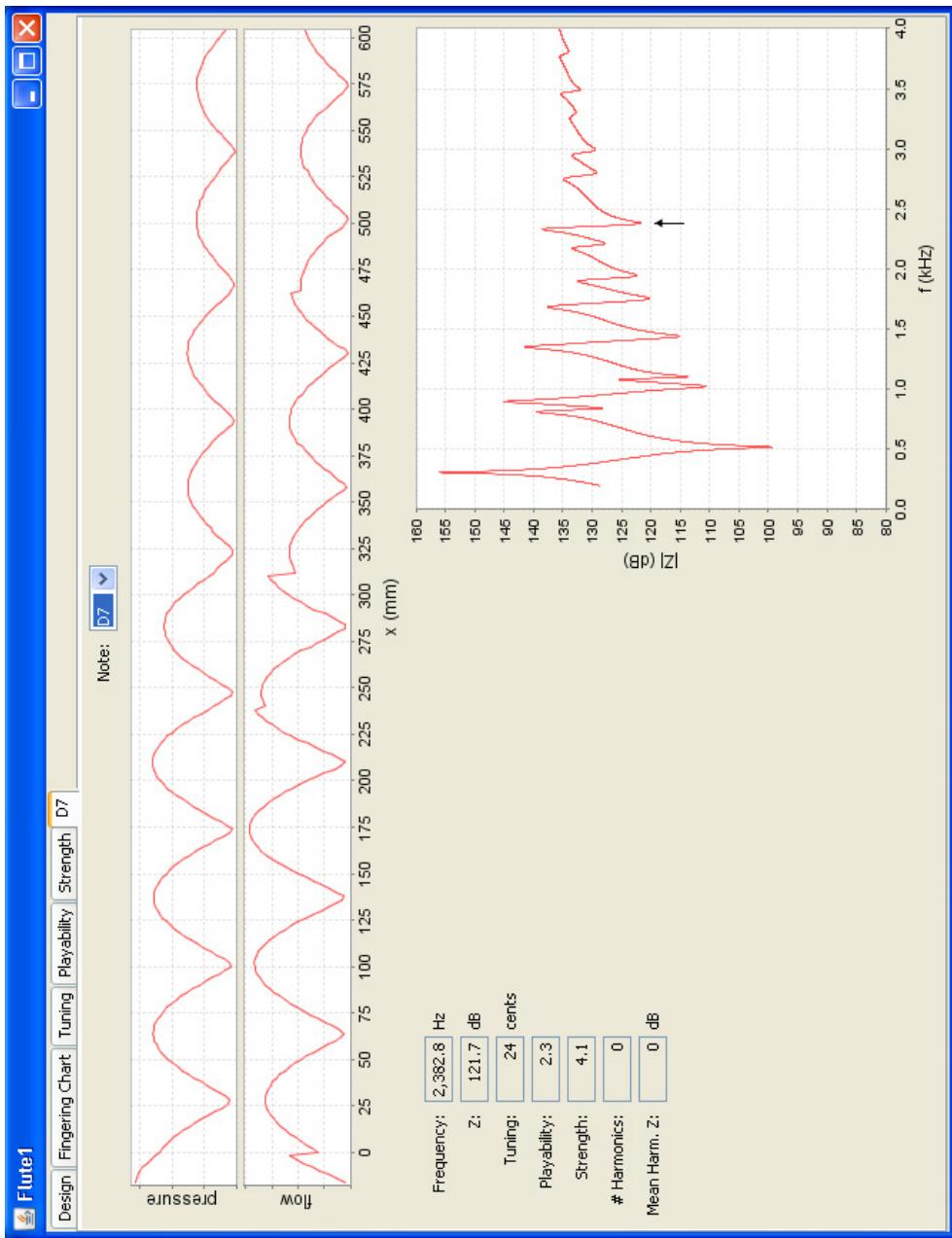


Figure 8.6: A screen shot of the note display tab of the user interface. The note D7 is selected.

Chapter IX

Applications and further directions

In this chapter several possible applications of this thesis are discussed and further work is suggested. Two specific applications of the software described in Chapter 8 are considered in detail as a guide to how similar problems may be approached. One application considered is adding a new hole to the modern flute, and the other is an analysis of an unplayable 19th century instrument.

9.1 ADDING A NEW HOLE TO THE MODERN FLUTE

The C♯ key on the modern flute performs many functions and its size and placement are chosen so as to perform each adequately. Consequently, some makers have added an extra hole and key. The problem is summarised in the following quote by American flutist and composer Robert Dick (Dick 1999):

The small C♯ hole is the most multi-purpose of any on the flute. There is no way that one hole can fulfill all of its functions perfectly. In balancing the roles of that hole—to make the second and third octave C♯s and to vent the second octave D and D♯, the third octave D, G♯, A and A♯, and the fourth octave C♯ and D—all designers have compromised. A full sized tonehole placed further down the flute would make wonderful C♯s but the vented notes would be terrible. A hole considerably smaller than the C♯ holes we're used to seeing would be better for venting. It would be placed visibly higher than the usual C♯ position. While making the vented notes nicer, it would produce extremely sharp and thin sounding C♯s.

Boehm goes into depth about this dilemma [*sic*] and his thought process about it in his book “The Flute and Flute Playing”. He devised the compromise small hole, accepting that it would do all of its jobs imperfectly, but well enough for musical players to correct, and he opted away from making the mechanism more complex. A more modern solution has been to add the key that most of us would call the “C♯ trill”, a full sized tone hole which is normally closed and opened by a touch operated by the right hand forefinger. This gives acoustically correct C♯s. Some folks don't care for the “extra” mechanism; I think it's great. Going even further, Alexander Murray, John Coltman and Jacques Zoon have devised mechanisms that switch between small and large C♯ holes depending on the note being played.

Note that the fingering used for third octave A♯ (A♯6) in this chapter and in Appendix A does not have an open C♯ hole. In this section I will use FluteCAD to decide on the best size and position of an extra C♯ hole in an attempt to improve the tuning of the Pearl modern flute (PF-661).

9.1.1 Unmodified tuning

The tuning of the unmodified Pearl flute is shown in Figure 9.1. The stopper was at 17.5 mm from the centre of the embouchure hole and the slide was open 8 mm. The tuning problems

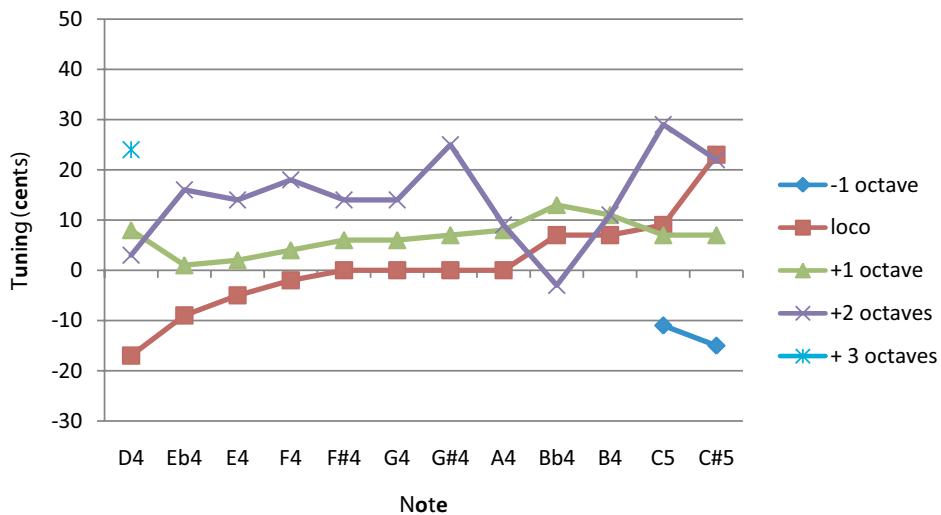


Figure 9.1: The predicted tuning of the unmodified Pearl modern flute.

due to the C♯ hole are immediately apparent—both C♯5 and C♯7 are over 20 cents sharp. Interestingly, C♯6 is no sharper than its adjacent notes; this was also seen in the played pitch measurements of Chapter 7. On the unmodified flute the C♯ key is 7.0 mm in diameter and is 238.4 mm from the embouchure hole.

The following sections describe the effects on tuning of simple changes to the software model of the flute.

9.1.2 An ‘acoustically correct’ C♯ hole

The first change I made was to increase the size of the C♯ hole and to place it at its ‘acoustically correct’ position. The advantage of this change is that the notes made with this hole have timbre more similar to other notes in the scale. The hole diameter was changed from 7.0 to 13.2 mm and the key diameter was changed from 13.0 to 19.0 mm (these values were chosen so as to match the properties of the next few holes). The height of the key above the hole was changed from 1.9 to 2.4 mm and the thickness of the key was changed from 3.6 to 4.0 mm (these are average values for the next few holes). The hole was moved 15 mm down the flute, making it 253.4 mm from the embouchure hole (at this position the consecutive distances between the first few holes form a geometric series). The resulting tuning is shown in Figure 9.2. Small changes in tuning can be seen for notes made with the C♯ hole closed: most notes are flattened, but the change is at most 4 cents. A few such notes are sharpened, but by no more than 1 cent. This change is consistent with an increase in the size of the cavity formed by the closed C♯ hole. For notes made with the C♯ hole open, C♯5, C♯6 and C♯7 are more in tune, although the spread in tuning between the three notes is nearly 20 cents. The tuning of D5 and D6 is acceptable, but E♭5 is over 40 cents sharp and D7 is out of range. These notes use the C♯ hole as a register hole and for this purpose the hole is too big and rather inefficient. The remaining notes that use the open C♯ hole (G♯6 and A6) are within a few cents in tuning compared to the adjacent notes). D♯5 and A6 have low playability compared to adjacent notes—likely due to the inefficiency of

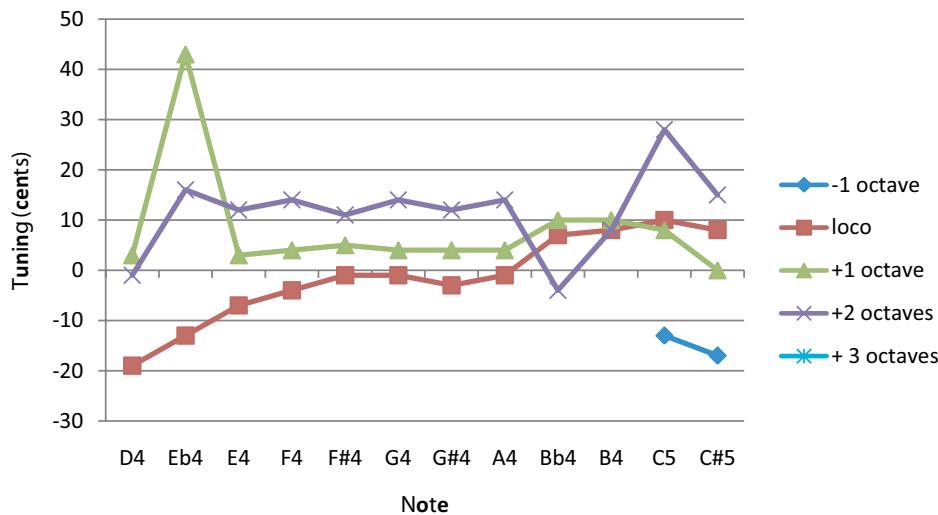


Figure 9.2: The predicted tuning of the Pearl modern flute after moving and resizing the C♯ hole.

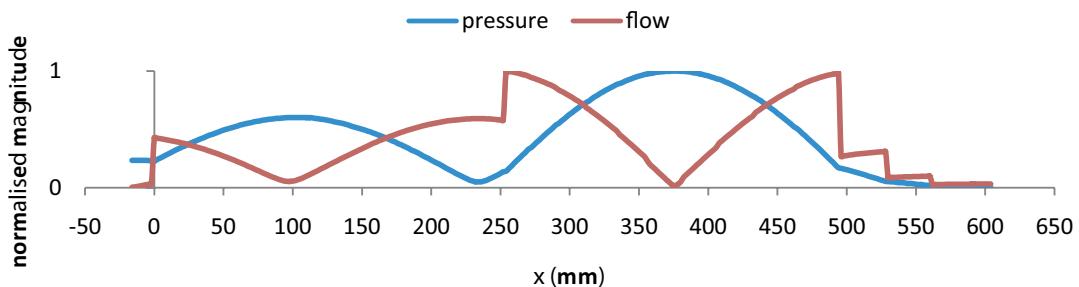


Figure 9.3: Normalised magnitude of pressure and volume flow along the flute for the note Eb5.

the C♯ hole acting as a register hole.

The pressure and flow profiles along the flute for the note Eb5 are shown in Figure 9.3. For this note the strongest resonance in the flute is a half-wave resonance between the open C♯ hole at 253.4 mm and the open Eb hole at 494.3 mm. The C♯ hole is not strictly acting as a register hole, since a register hole should detune the lower resonance (Eb5 in this case) and leave the desired resonance unchanged. The hole is too big and too far down the flute to satisfy this second requirement.

The modified C♯ hole is evidently not a very good register hole. So perhaps we could do without it. If the C♯ hole is closed for all fingerings except C♯5 and C♯6, the tuning shown in Figure 9.4 results. Now G♯6 and C♯7 are approx. 25 cents flat (relative to adjacent notes) and the other notes that ordinarily use the C♯ hole as a register hole are in tune. The C♯ hole acts as more than just a register key for the notes G♯6 and C♯7: it also sharpens them. This role must be taken into account in deciding on the placement of a new ‘register’ hole.

All of the fingerings that use the C♯ key as a ‘register hole’ or a ‘vent hole’ were analysed for their pressure node nearest the C♯ key position. (The terms ‘register hole’ and ‘vent hole’ are rather loose, but are understood here to refer to any open hole that does not act primarily

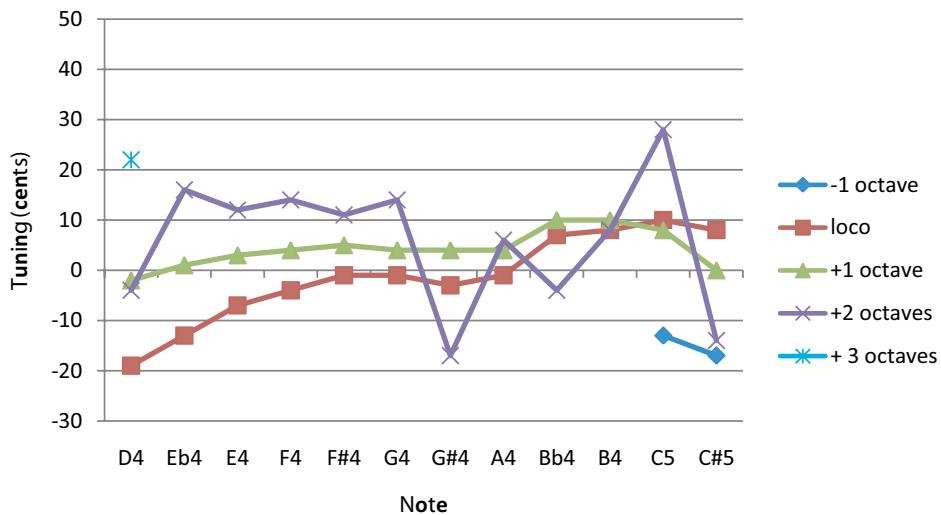


Figure 9.4: The predicted tuning of the Pearl modern flute after moving and resizing the C♯ hole and closing the hole for all notes except C♯5 and C♯6.

as an acoustic short circuit.) The nodal positions for these notes are shown in Figure 9.5. The average nodal position was 253.2 mm from the embouchure hole, almost exactly the position of the new larger C♯ key. As discussed above, however, the ‘register hole’ also serves to sharpen the notes G♯6 and C♯7, and must be placed somewhat higher than the new ‘acoustically correct’ C♯ key in order to fulfil this function.

9.1.3 Adding a separate register hole

I next added a vent hole at 239.5 mm from the embouchure hole. The hole was made 4.0 mm in diameter with similar key dimensions to the original C♯ key. The fingerings for all notes previously using the C♯ key except for C♯5 and C♯6 were changed to use this vent key instead. The resulting tuning is shown in Figure 9.6. The use of this vent hole brings G♯6 and C♯7 back into tune, without adversely affecting the tuning of the other notes that use this key. D7 is still sharp, but this note was sharp in the original flute and none of the modifications made brought this note into tune. This note is near the top of the range on the modern flute and at such a high frequency the register hole has little effect. This note is insensitive to changes in the register hole and may need to be dealt with separately (The Virtual Flute may be used to search for other fingerings). The exact size of this vent hole is not critical, although in general a smaller vent hole is preferred since it perturbs the tuning of notes less when not at an optimal position. The size of 4.0 mm was chosen by examining the relevant impedance spectra and ensuring sure that any lower resonances are detuned and weakened sufficiently by opening the vent hole. Compare Figure 9.6 with Figure 9.1 to see the improvements in tuning achieved as a result of the minor changes made in this section.

9.1.4 Remaining tuning problems

The tuning shown in Figure 9.6 shows some octave stretching, with the notes in the range D6 to C♯7 being more than 10 cents sharper than the range two octaves lower. As discussed in

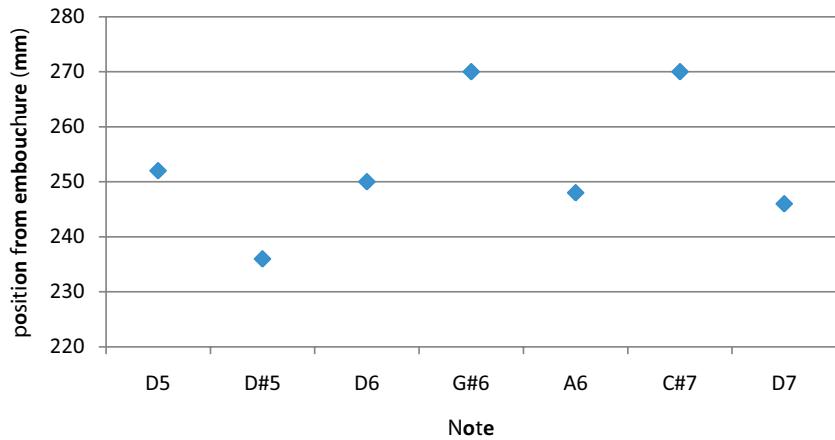


Figure 9.5: Position of pressure nodes nearest the C \sharp hole (measured from the centre of the embouchure hole) for all notes that use the C \sharp hole other than C \sharp 5 and C \sharp 6. The C \sharp hole was closed for this calculation.

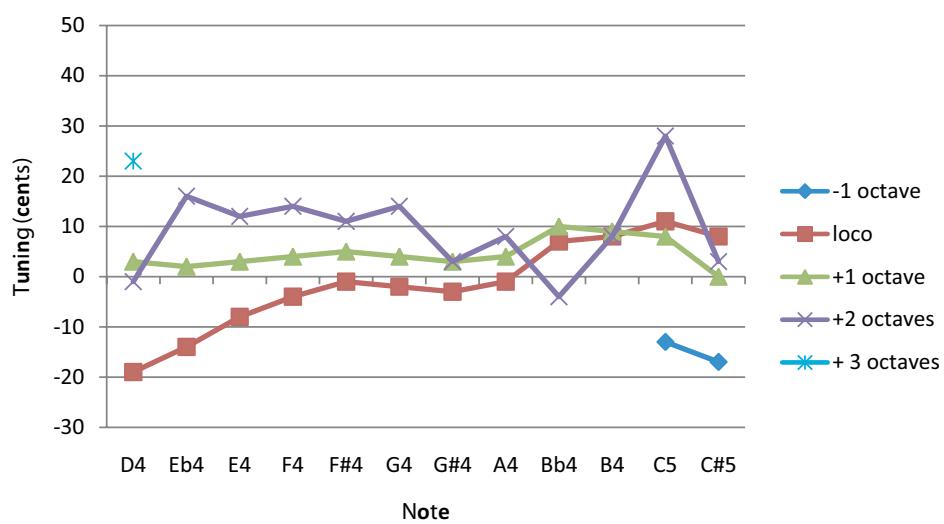


Figure 9.6: The predicted tuning of the Pearl modern flute after moving and resizing the C \sharp hole and adding a new register hole.

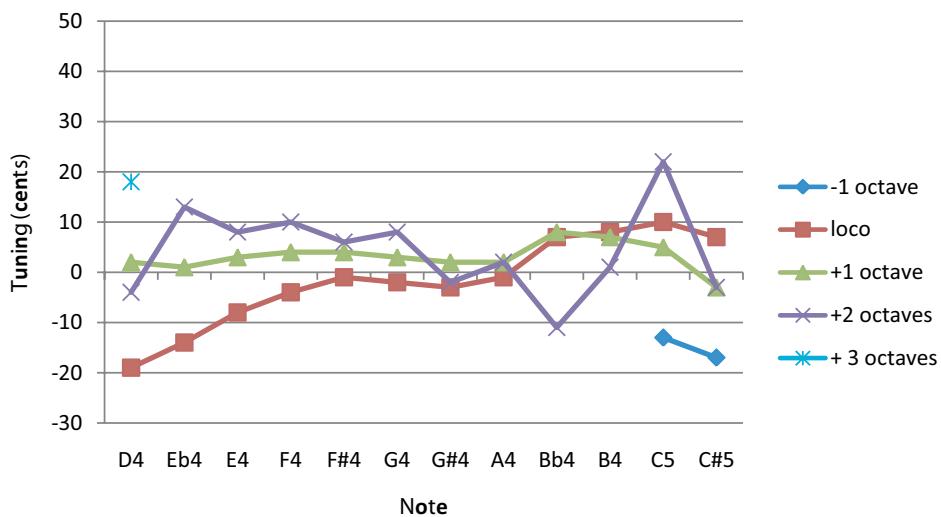


Figure 9.7: The predicted tuning of the modified Pearl modern flute after adjusting the stopper position.

Chapter 7, some octave stretching is preferred by musicians (Ward 1954) which may explain this tendency. By changing the stopper position from 17.5 to 18.5 mm, the octaves can be reduced, as shown in Figure 9.7.

The tuning shown in Figure 9.7 is clearly not optimised. The first few notes are flat, over the first octave the notes become increasingly sharp and notes such as Bb6 and C7 differ significantly in tuning from adjacent notes. Some of these tuning problems may be able to be addressed by adjusting the regulation of the flute, and others may require small changes to the sizes and positions of the holes. Whether such an ‘optimised’ flute would be judged by players to be ‘in tune’ is not entirely clear—since good flute players are adept at making subtle and semi-unconscious adjustments to their embouchure to make up for the short-comings of existing instruments. Further, flutes with slightly stretched octaves may be preferred. A possible methodology to be followed in any attempt to improve the modern flute has been illustrated by the simple example given in this section.

9.2 EIGHT-KEY FLUTE BY RUDALL & ROSE

A 19th century flute in the collection of the Powerhouse Museum in Sydney was analysed using FluteCAD. The software model for the flute is based on geometrical measurements made by Terry McGee. The flute is by London makers Rudall & Rose and dates from 1838–1847. The makers’ serial number for the flute is 3522 and the Powerhouse Museum catalogue number is 93/117/5. The flute is made of wood and metal, and has a ‘patented adjustable metal-lined headjoint’. This headjoint has a screw cap that moves both the tuning slide and the stopper, but at different rates (see <http://www.mcgee-flutes.com/patent.html>). The flute has splits to the wood of the head and barrel (almost universal in flutes from this period, since the wood tends to shrink around the metal lining) and for this reason and its historical worth the flute is unplayable. A photograph of the flute is shown in Figure 9.8.



Figure 9.8: Eight-key flute by Rudall & Rose (3522). Photographs courtesy of the Powerhouse Museum, Sydney. Used by permission.

Table 9.1: The relationship between the slide extension and the position of the stopper relative to the embouchure hole.

Slide extension (mm)	Stopper position (mm)
0.0	20.0
10.0	19.0
20.0	17.0
31.0	15.5

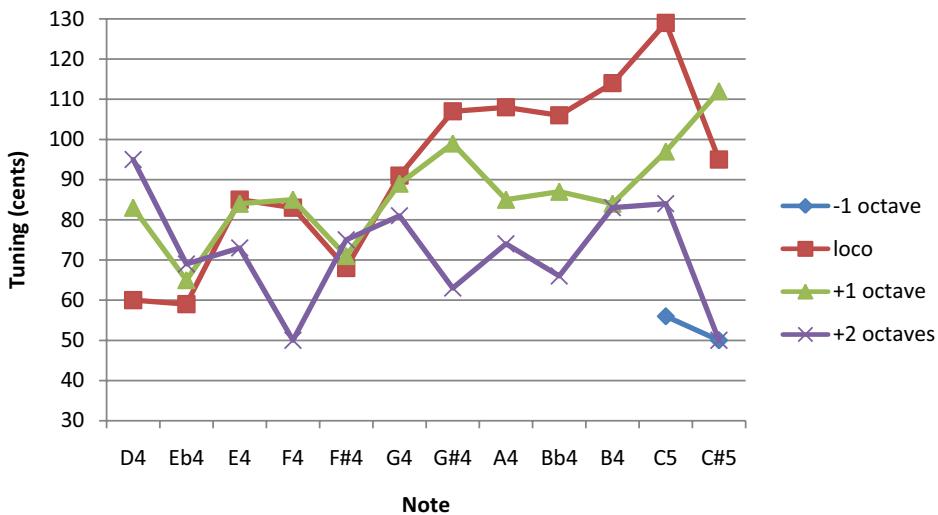


Figure 9.9: The predicted tuning of the Rudall & Rose flute with the tuning slide at 0.0 mm and the stopper at 20.0 mm.

Terry McGee measured the size and position of the finger holes on this flute, but was unable to obtain precise measurements of the keyed holes. This is because the pins on old flutes are often corroded in, and the wood is shrunken, making the removal of the pins difficult and risky. The diameters of the keyed holes were therefore taken from a similar flute in private hands. The finger holes are all undercut conically, although no attempt was made to quantify the extent of undercutting. The top half of the bore of the left hand body joint was ovoid in shape, so the maximum and minimum diameters were measured and averaged in the model. The tapered elliptical embouchure hole was approximated by a cone with the same entry and exit areas.

This flute has a particularly long tuning slide with a maximum extension of 31 mm. Due to the ‘patent head’ at the maximum slide extension the stopper is 15.5 mm from the embouchure hole. The positions of the stopper with the slide fully closed and at intermediate positions are given in Table 9.1.

FluteCAD was used to predict the tuning of the Rudall & Rose flute in each of the configurations listed in Table 9.1. The results are shown in Figures 9.9–9.12. In each graph the tuning is given in cents relative to the equal tempered scale with the note A4 at 440 Hz. The mean tuning and standard deviation for each of the slide positions is shown in Figure 9.13.

The flute has the best internal tuning (smallest standard deviation) with the slide at 10 mm and the stopper at 19 mm, however the tuning is far from optimised even in this configuration

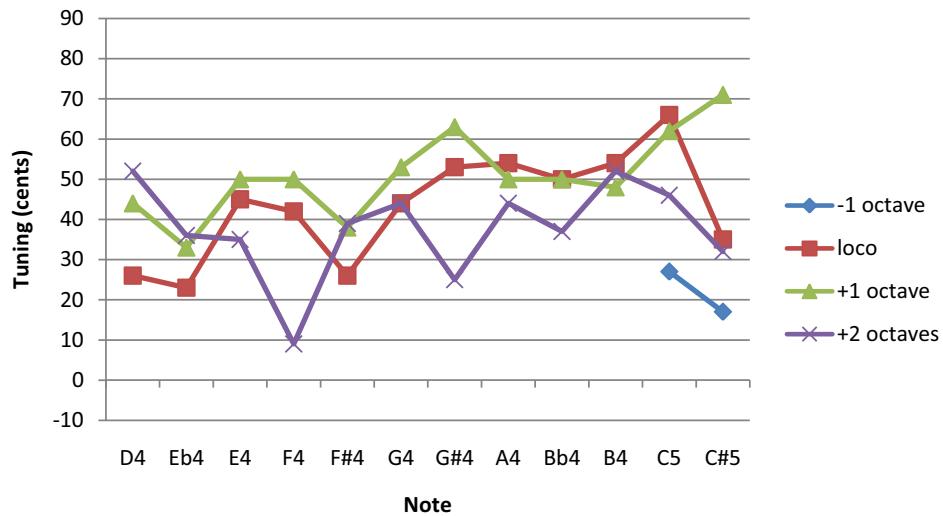


Figure 9.10: The predicted tuning of the Rudall & Rose flute with the tuning slide at 10.0 mm and the stopper at 19.0 mm.

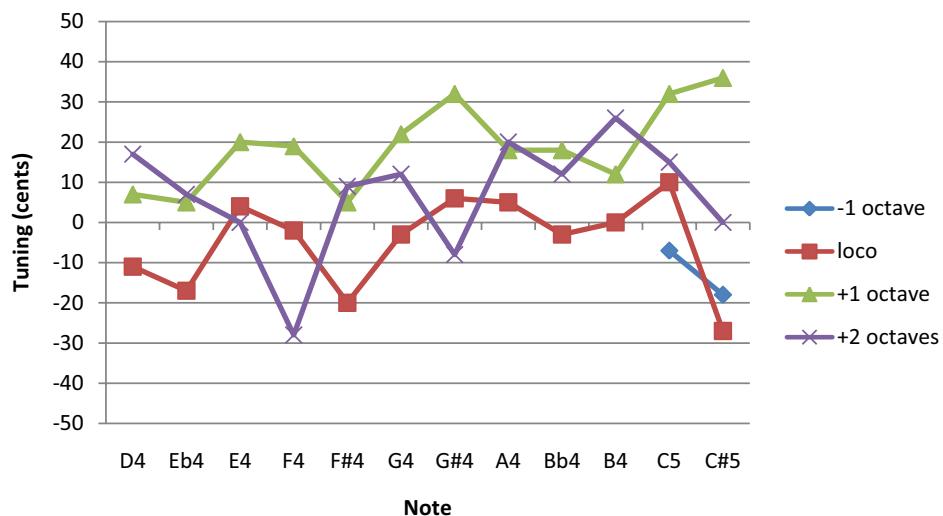


Figure 9.11: The predicted tuning of the Rudall & Rose flute with the tuning slide at 20.0 mm and the stopper at 17.0 mm.

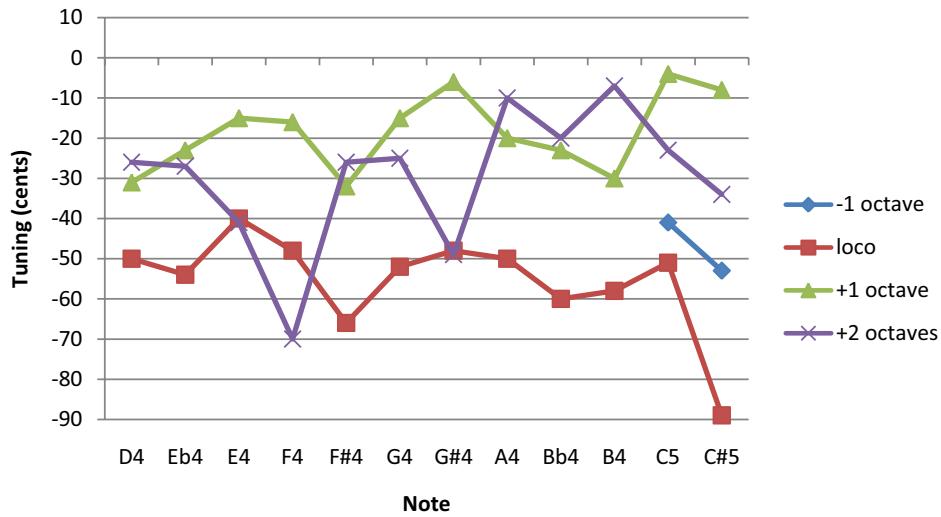


Figure 9.12: The predicted tuning of the Rudall & Rose flute with the tuning slide at 31.0 mm and the stopper at 15.5 mm.

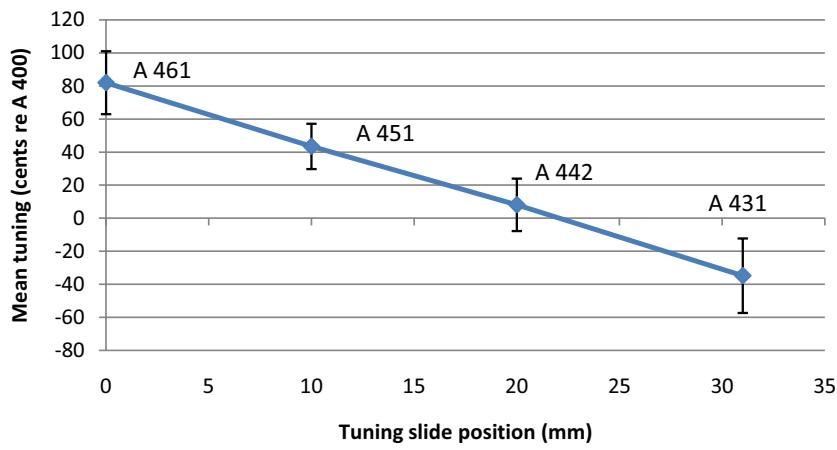


Figure 9.13: The mean tuning (relative to A 440) and standard deviation of the Rudall & Rose flute for each of the tuning slide and associated stopper positions given in Table 9.1. The approx. tuning of A4 required for the flute to be in tune overall is printed next to each data point.

(i.e. there is no slide position and—by inference—no design pitch where the flute is perfectly in tune). In most flutes the stopper and the tuning slide are not coupled, and many flutists (perhaps the majority) tend to adjust the tuning slide regularly but leave the stopper position unchanged. For small changes in pitch this is perfectly acceptable, since flute tuning is much more sensitive to the slide position than to the stopper position. However, for large changes in slide position the tuning can be significantly improved by concomitant changes in the stopper position. Many different pitches were in use in the 19th century and flutes were designed to play adequately (if not well) over a wide pitch range. For a detailed discussion of the many pitches in use in the 19th century, see Terry McGee's web site <<http://www.mcgee-flutes.com/>>. The following comments of Terry's (shown in italics) explain the problem of different pitches, and how various makers tried to accommodate them.

Before the 19th century, pitch was considerably lower than the modern standard pitch in which the note A4 is assigned a frequency of 440 Hz (denoted by the shorthand A 440). 17th century French flutes work well around A 392 (a whole tone lower in pitch), and A 415 is taken as a convenient pitch for the late baroque (18th century). In France and Germany, pitch had stabilised by the early 19th century to A 435, but in England, where this flute is from, a wide range of pitches were in use.

A flute by Richard Potter in the Bate collection in England is a good example of how this was first dealt with. Dated at 1782, it has three interchangeable left-hand sections (corps de rechange), giving pitches estimated as A 418, A 427 and A 436. The three bodies, which differ in length by a total of 18 mm, are marked '4', '5' and '6' to denote their pitches. Not long after, Potter introduced the tuning slide as a cheaper way to achieve the necessary tuning range. He continued the tradition of denoting the pitches with incised rings on the slide, also marked '4', '5' and '6'. Conscious of the fact that the stopper should be moved to get the best results at these differing extensions of the slide, he had an indicator rod sticking through the cap, marked with corresponding rings.

A professional musician might encounter at least three different pitches in a day: e.g. an older organ in a church tuned say to A 410, a domestic piano at an afternoon tea tuned to just over A 430 (actually C 256), and a Philharmonic Orchestra concert at a pitch as high as A 455. Attempts were made by the Society of Arts to compromise these to A 445, but the Philharmonic movement stuck to their guns until 1895, when they suddenly accepted 439 Hz (only marginally higher than the Continental A 435). 439 being a prime number was dropped in favour of 440 at a later international conference on the basis that it could not easily be generated electrically.

So, the tuning slide had to deal with a range of at least 25 Hz (the difference between the Philharmonic orchestra at A 455 and the domestic piano at A 430) and maybe more. According to the flutemakers' rule-of-thumb that a pitch change of 1 Hz corresponds to a change in length of 1 mm, this implies a movement of the slide of at least 25 mm. Apart from adjusting the slide to attempt to encompass these wide ranging pitches, a corresponding change was needed to the stopper to restore, as well as can be achieved, the internal tuning of the instrument.

In 1832, Rudall & Rose patented a mechanised way of keeping the stopper and slide in their optimum relationship, so by then all our the player had to do was wind the stopper cap for best

overall tuning. This is the Patent Head seen in the Rudall & Rose #3522. The 31 mm range of the slide encompasses the 25 mm range necessary to play in varied pitches, with some allowance for players who naturally tend sharper or flatter than average.

Clearly, none of the tuning graphs shown in Figures 9.9 to 9.12 shows perfect tuning. Some of the tuning anomalies may be due to our incomplete measurement of the hole and key dimensions or because the model in its current form does not take account of undercutting. However, the predicted tuning is for the most part consistent with the tuning of similar playable flutes. Notably, F \sharp 4 and F \sharp 5 are between 10 and 20 cents flat compared to nearby notes, a common feature of small-holed classical flutes like this one. (Flat F \sharp s are a direct consequence of a compromise in hole placement needed because of the limited stretch of the fingers on the right hand. The maker Siccama overcame this problem, but needed to introduce keys for the third fingers of each hand.)

Classical flute makers are often asked to copy extant 19th century instruments (which may or may not be playable) while correcting where possible for tuning errors. They may also change the pitch so that optimal internal tuning is achieved at say A 440. Indeed modern makers are at a distinct advantage since the flute need only play over a relatively small range of pitches. FluteCAD can assist greatly in such designs. The tuning of the Rudall & Rose flute could be optimised, but the parameters are heavily constrained by human physiology and various aesthetic decisions of the maker (e.g. the number of keyed holes, and how far he is willing to deviate from the hole dimensions of the historical instrument). The required changes would be subtle and numerous, and likely of interest only to flute makers and the truly dedicated reader—such a task has not been attempted for this thesis.

9.3 CONCLUSIONS AND FURTHER DIRECTIONS

In this thesis, a computer program was developed for the acoustical design and analysis of flutes. Since the practical goal of the project was to provide a tool for flute makers and flute historians, the provision of a user-friendly graphical interface was deemed of great importance. The majority of potential users of the program have little or no knowledge of physics or acoustics and so the interface should be as devoid as possible of jargon and esoteric detail. The resulting interface serves these purposes well, and the provision of pressure and flow profiles for each note are of great help to instrument makers. Of course the ultimate appraisal of the software is the prerogative of users. The software has been well received so far, and with ongoing collaboration minor omissions and oversights will be remedied.

The underlying physical model of the flute is a one-dimensional waveguide model fitted through the use of semi-empirical parameters to a database of measured flute impedance spectra. The impedance database itself is a valuable resource and is (we believe) accurate enough to make reliable musical predictions. Such accuracy is a consequence of the careful calibration and optimisation techniques developed and outlined in Chapter 3, building on the considerable expertise of the Music Acoustics Laboratory (Wolfe et al. 2001a, Smith et al. 1997). A database of clarinet impedance spectra was also made, and this is currently being used, by another student, to develop a ‘Virtual Clarinet’.

In order to model tone holes drilled directly through the wall of a woodwind instrument and stopped by the player's fingers, the lumped-element impedances for such holes were measured over a wide range of parameters, and fit-formulae were derived for the associated length-corrections (i.e. for the reactive part of each measured impedance) (Chapter 4). Such formulae for closed finger holes were not found in an extensive literature search, but they are important since the player's finger can have a large impact on the length corrections for closed tone holes. The measurements were subject to some systematic errors (discussed in §4.4). These did not materially affect the determination of the length corrections, but did compromise the measurement of the resistive part of each tone hole impedance. In future, it may be beneficial to determine these equivalent-circuit impedances to a higher degree of precision, and to measure the resistance at tone-hole junctions. Some suggestions for how to improve these measurements are given in Chapter 4. It would also be of interest to investigate non-linear effects at finger holes. This has been done for open tone holes such as are found on the modern flute (Dalmont et al. 2002). The range of tone hole geometries studied does not include all tone hole types found on woodwind instruments, such as the long sloped holes found on bassoons or the holes on the clarinet lined with a metal pipe that protrudes into the bore. Undercut tone holes were also not measured. In future work, it is hoped that the database of tone hole impedances will be extended to encompass many of these geometries.

Further work is needed on the timber used to make classical flutes. In Chapter 6 the effects of humidity and oiling on pipes made from radiata pine were investigated. This study could be expanded to include typical flute timbers and to examine changes in the acoustic impedance and playing characteristics of a new flute over time. Players and makers speak of improvement in the performance of a new flute over time, as the flute is 'played in'. Is this due to gradual polishing of the bore with repeated oiling and swabbing? Do the sharp edges at the embouchure hole and tone holes slowly wear away, and could this improve the flute's performance? Or are there no appreciable changes at all in the acoustic properties of a flute in its first weeks and months of playing? Could it be the flutist, and not the flute, that adapts (plays in) over the first few weeks? Answers to some of these questions could lead to more informed methods of oiling for both makers and players. In this thesis the standard viscothermal theory for wall losses was used. It is noted that there has been no conclusive experimental validation of this theory, and in most practical situations the wall losses appear to be somewhat greater than predicted. It would thus also be interesting to use the impedance spectrometer to gauge the validity of the viscothermal model for wall losses. Deviations from theory may result for timber pipes, even those with dense grain and a well-sealed surface. A clearer understanding of wall losses would lead to more accurate impedance modelling.

In Chapter 7 the effect of the player on flute tuning is investigated and an empirical correction is derived and included in the impedance model of a played flute. No attempt was made to incorporate a physical model of the jet excitation, since the main goal of the thesis was to predict flute tunings which are largely determined by the frequency of the (empirically corrected) impedance minima. Further work in this area is warranted, for several reasons. Firstly, by combining the impedance model with a physical model of the jet excitation, one may synthesise

the sound of putative flute designs to get aural feedback on the tuning and timbre. Secondly, it is likely that the tuning of a note is not completely determined by the frequency of its impedance minimum, particularly when there are other minima in the impedance spectra near the harmonics of the note. If these do not coincide exactly in frequency then the frequency of the fundamental may be shifted. Experimental investigation of this possibility may allow for greater precision in tuning predictions. Thirdly, flute players in different traditions use very different embouchures. The corrections derived in Chapter 7 were based on the playing of three classically trained flutists. It would be interesting to widen this study to include players with a wider range of experience.

In its present form FluteCAD predicts the acoustics of flutes with a given design. It is up to the maker to use the information to improve such flutes. The next step in the development of the software is to include optimisation. Given a suitable set of constraints, procedures could be written to find the best possible tuning. Currently, the software is probably too slow for such optimisation, but perturbation theory could be used to quickly calculate the effect of incremental changes. It is also not trivial to add constraints to the flute design. For example, the finger holes on a flute cannot be moved relative to each other without considering the reach of the fingers. Also, changes in bore profile need to be machinable (this often means that the bore diameter of each joint should change monotonically over the length of the joint). However, with ongoing collaboration with flute makers, problems such as these may be overcome.

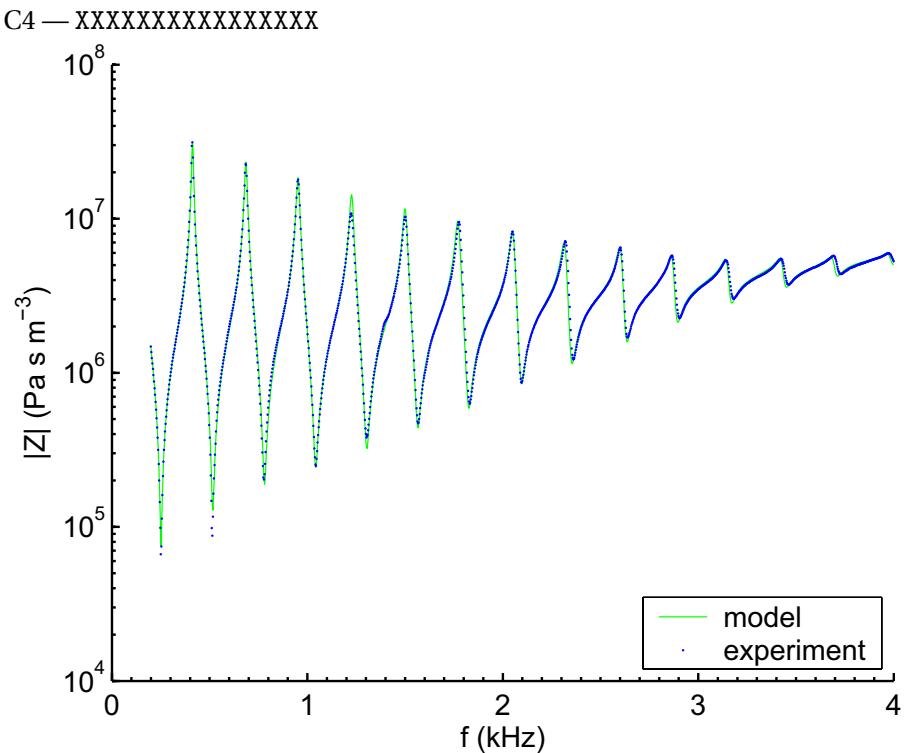
Appendix A

Impedance spectra

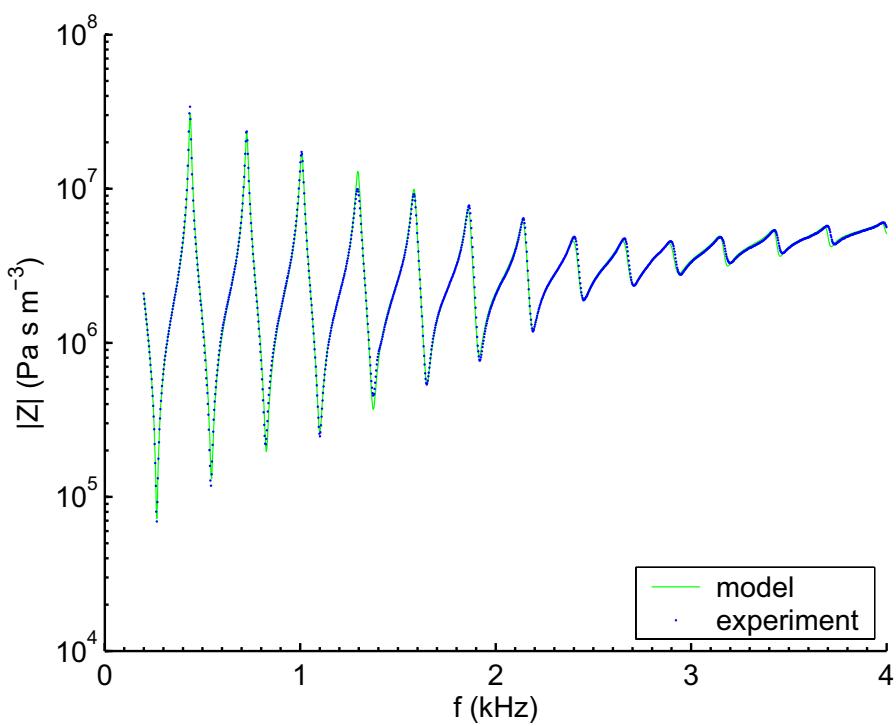
A.1 PEARL MODERN FLUTE

The following graphs show the magnitude of the measured input impedance spectra for a modern (Boehm) flute (Pearl PF-661, open hole with C foot) for all recommended fingerings. The tuning slide was set at 4 mm and the cork at 17.5 mm. The impedance spectra were measured using an impedance head of diameter 7.8 mm. The inertance of a stub of air of length 5 mm and diameter 7.8 mm was added to the measurements and simulations in order to approximate the effect of a player's face impedance and to facilitate comparison with earlier measurements (Wolfe et al. 2001a). The measurements were made at $T = 21.6 \pm 0.5^\circ\text{C}$ and relative humidity $37 \pm 1\%$.

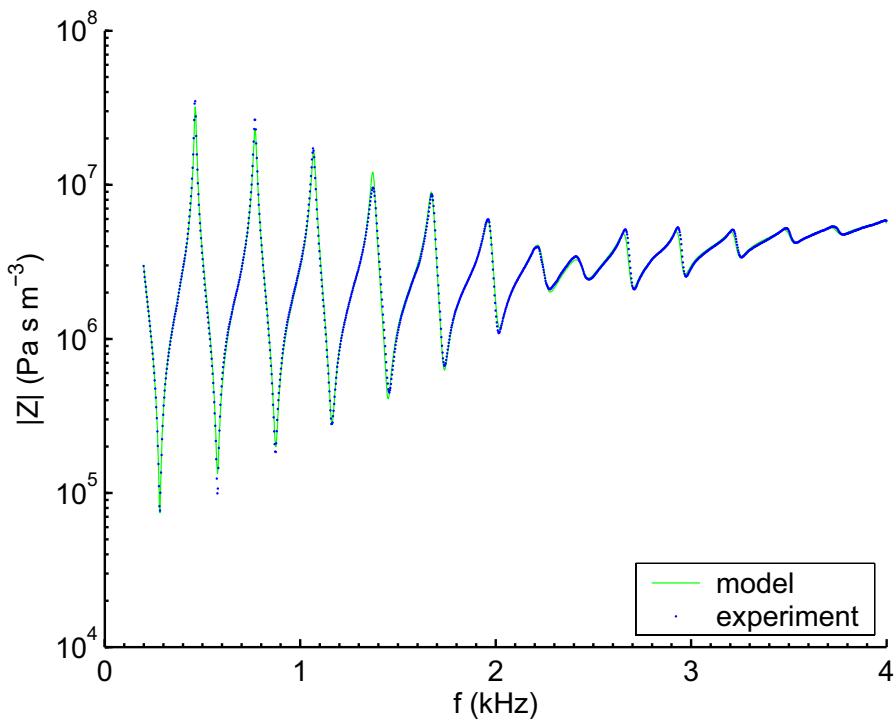
Measurements were also made with a B foot and for several alternate and mutiphonic fingerings. For brevity these are not reproduced here. For a complete set of measurements together with sound files and comments for each note see <<http://www.phys.unsw.edu.au/music/flute/>>.

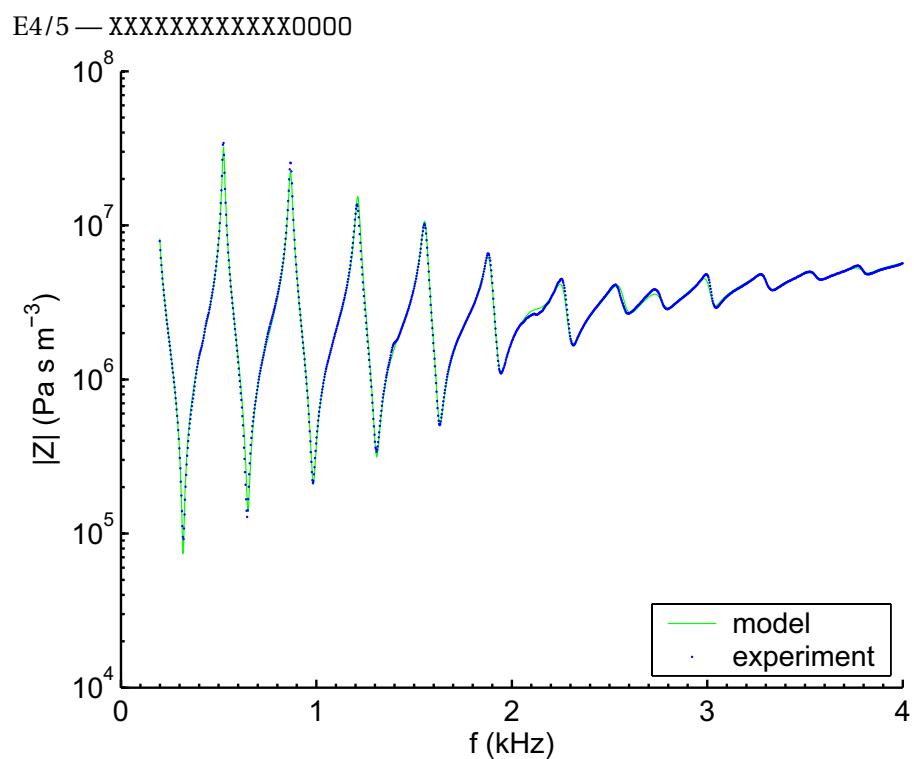
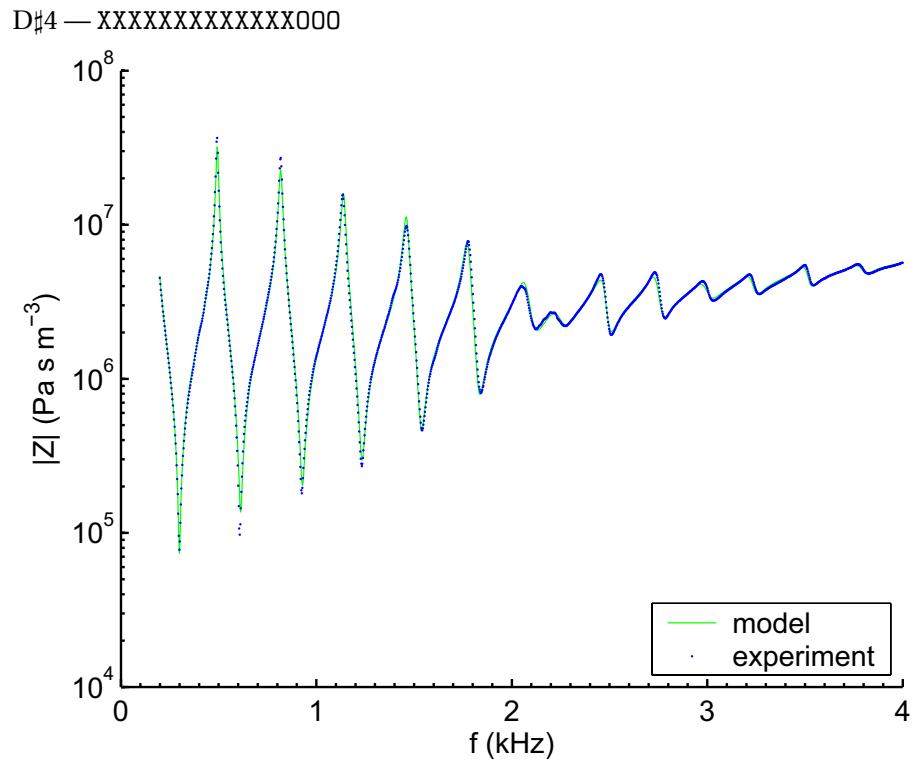


C‡4 — XXXXXXXXXXXXXXXX0

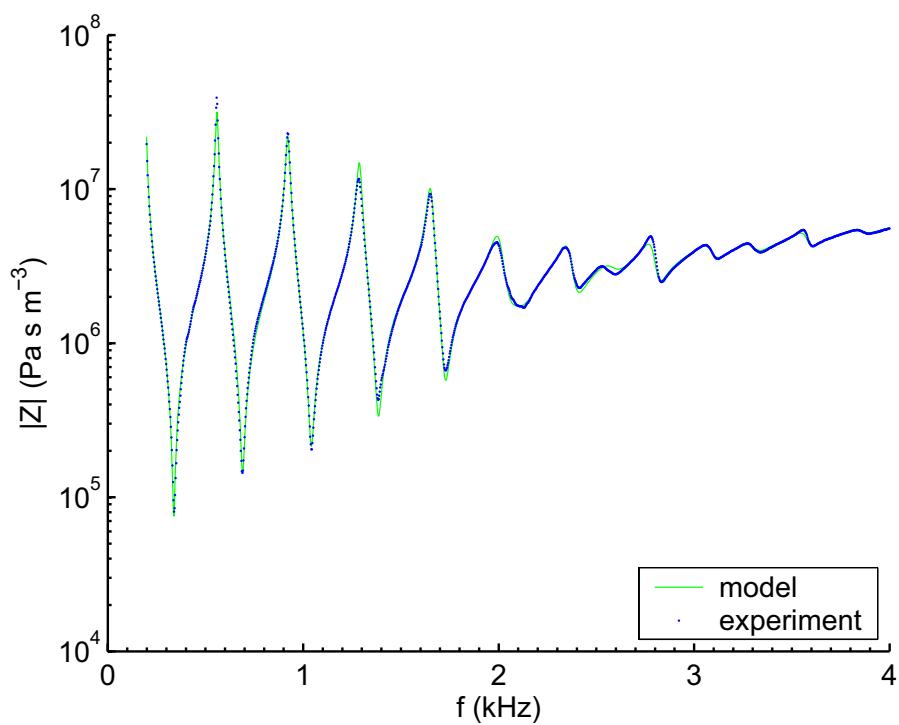


D4 — XXXXXXXXXXXXXXXX00

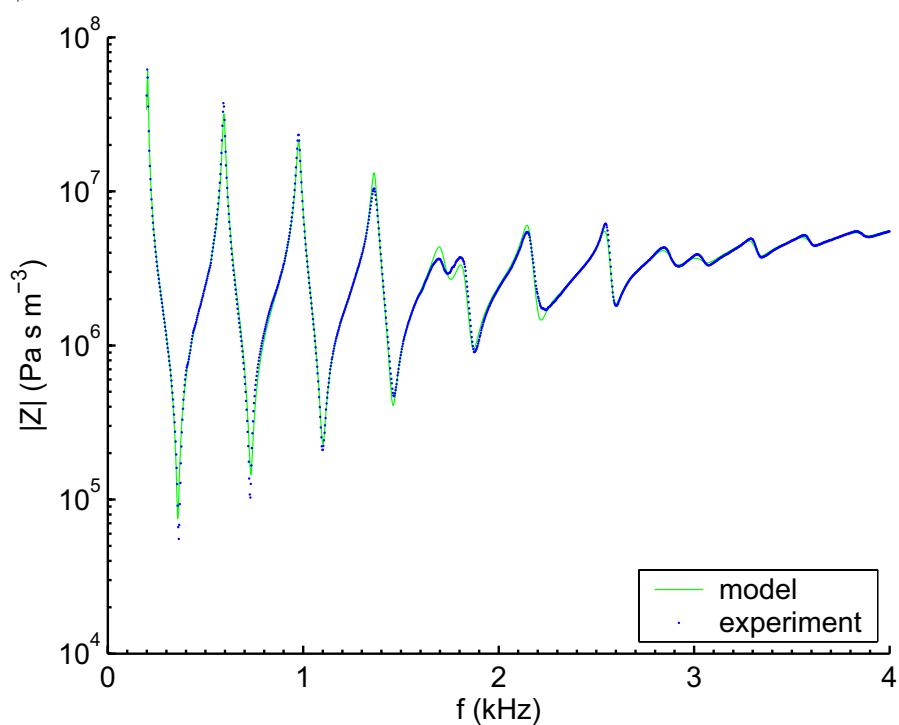


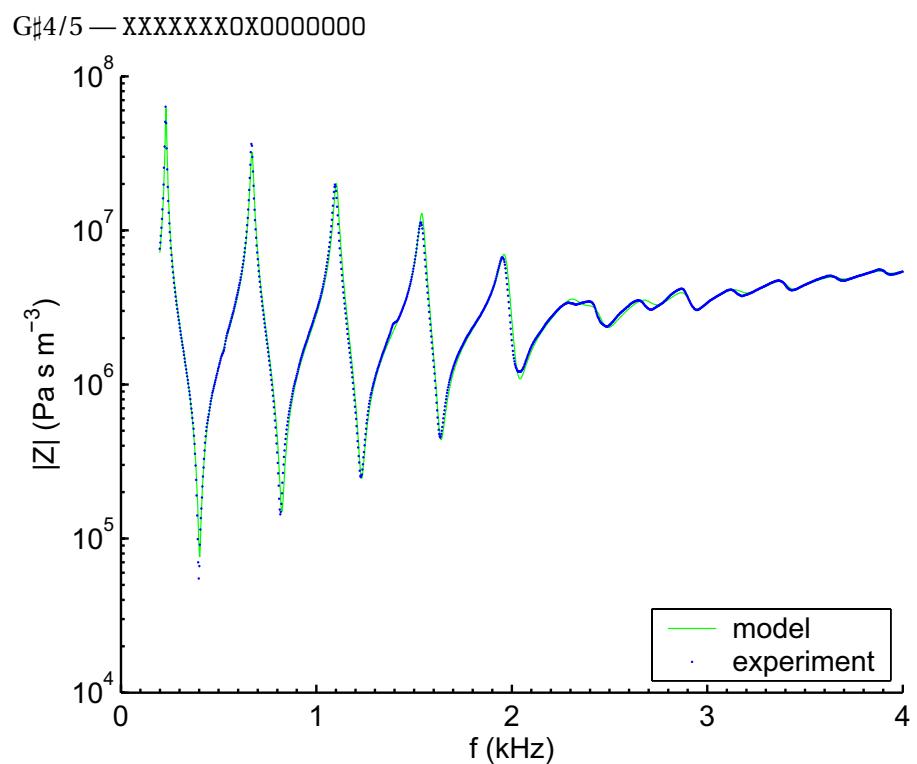
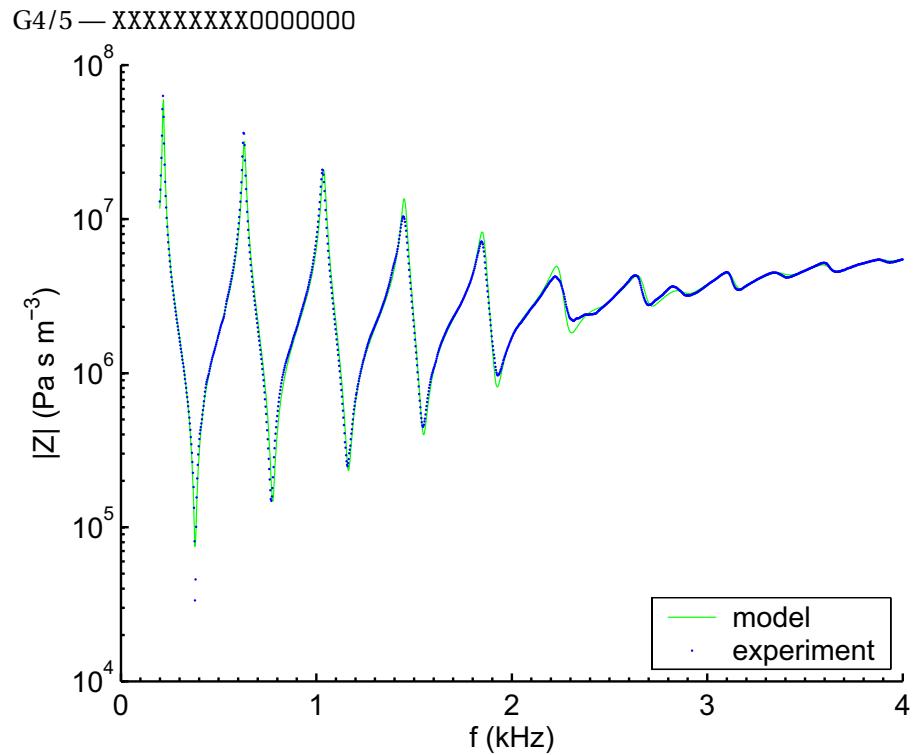


F4/5 — XXXXXXXXXXXXXXX00000

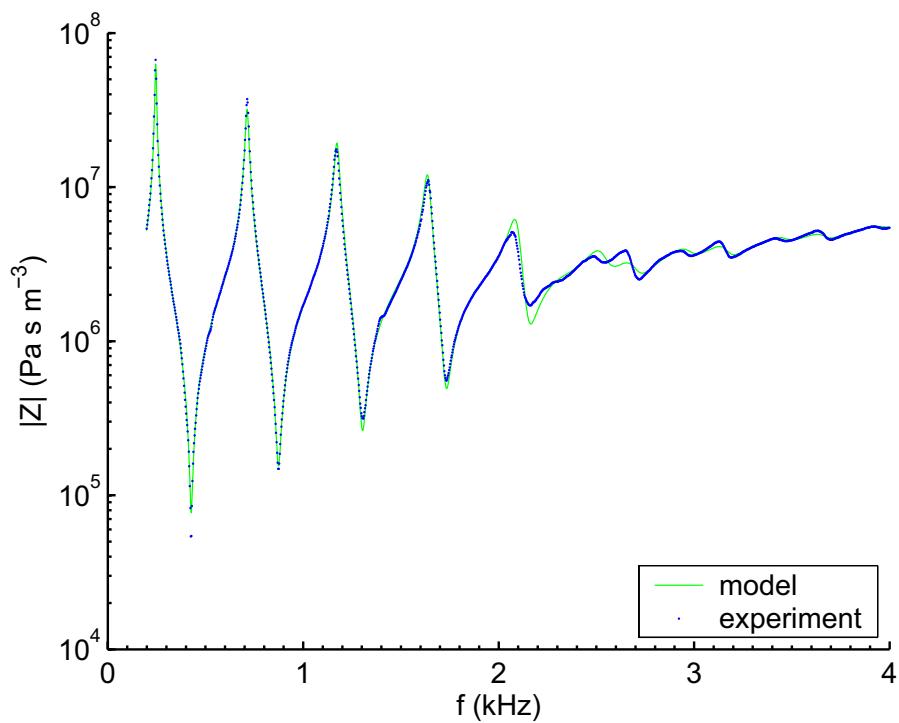


F#4/5 — XXXXXXXXXXXXXXX00X000

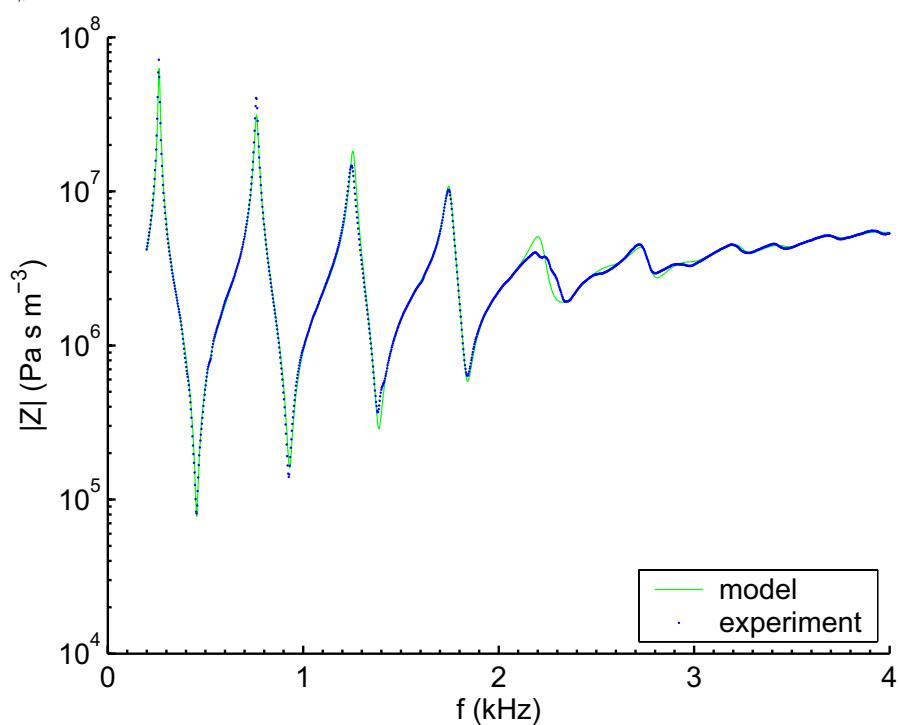


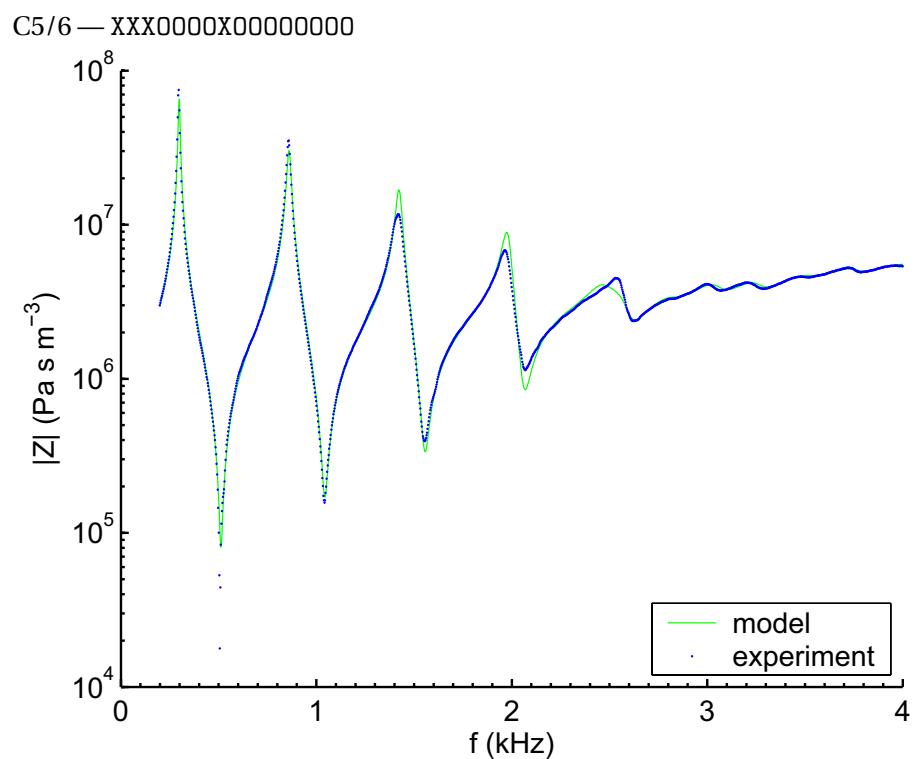
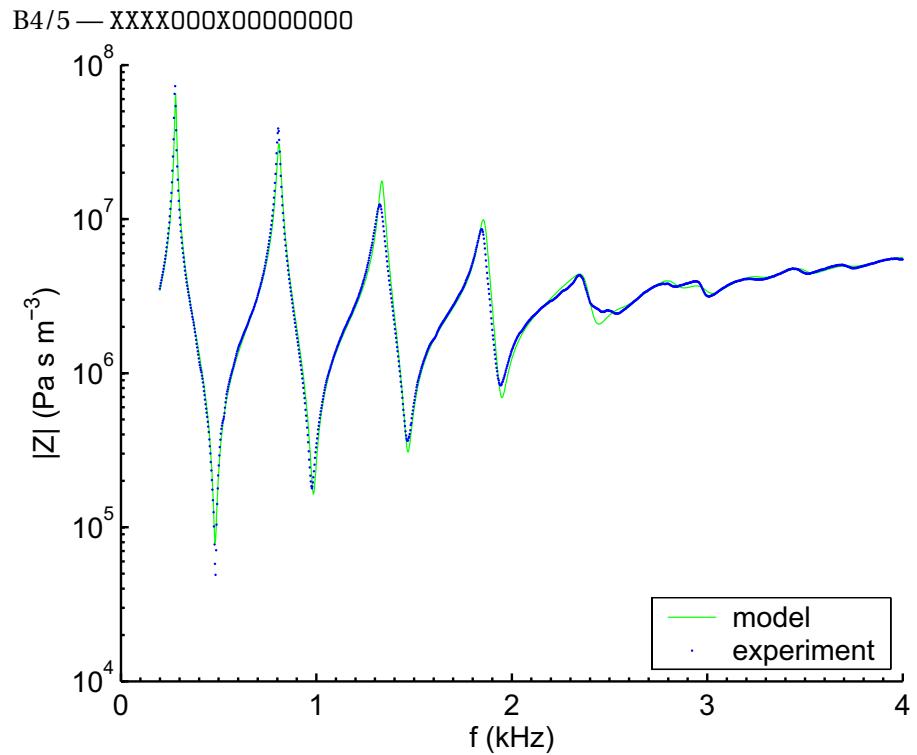


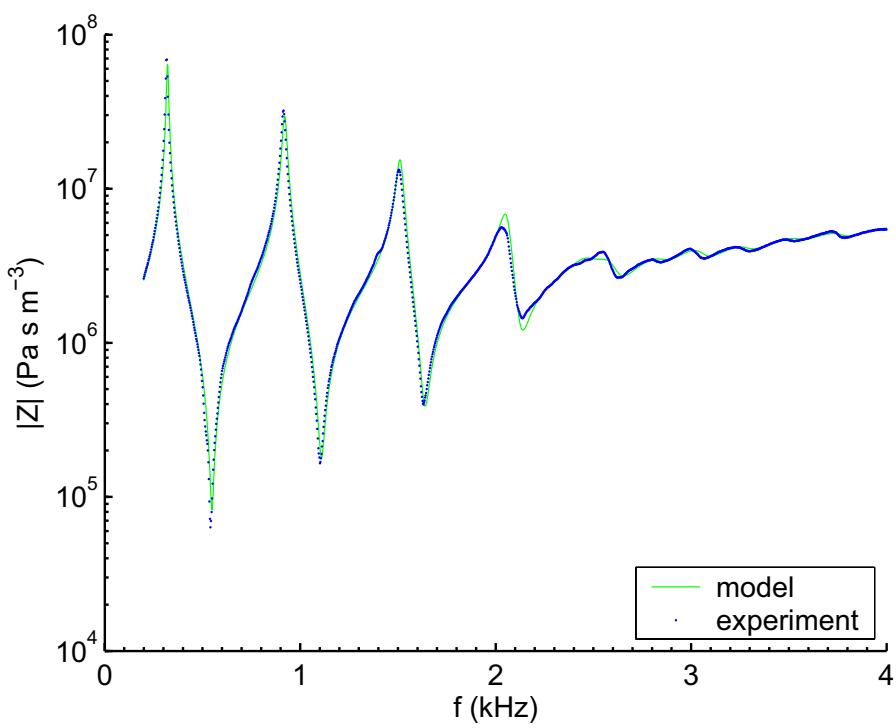
A4/5 — XXXXXOX000000000



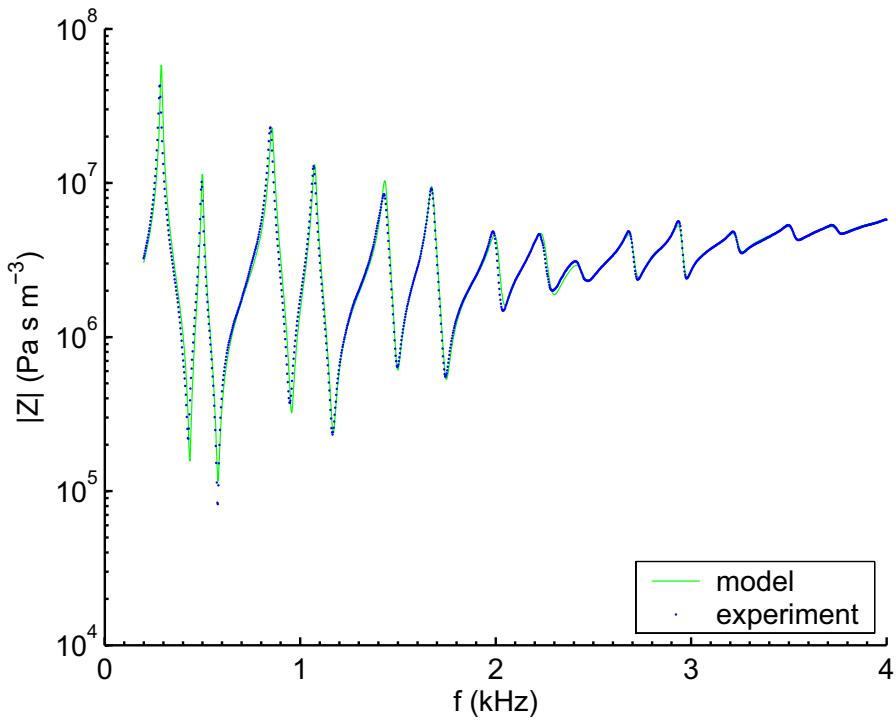
A4/5 — XXXXXOOX000000000

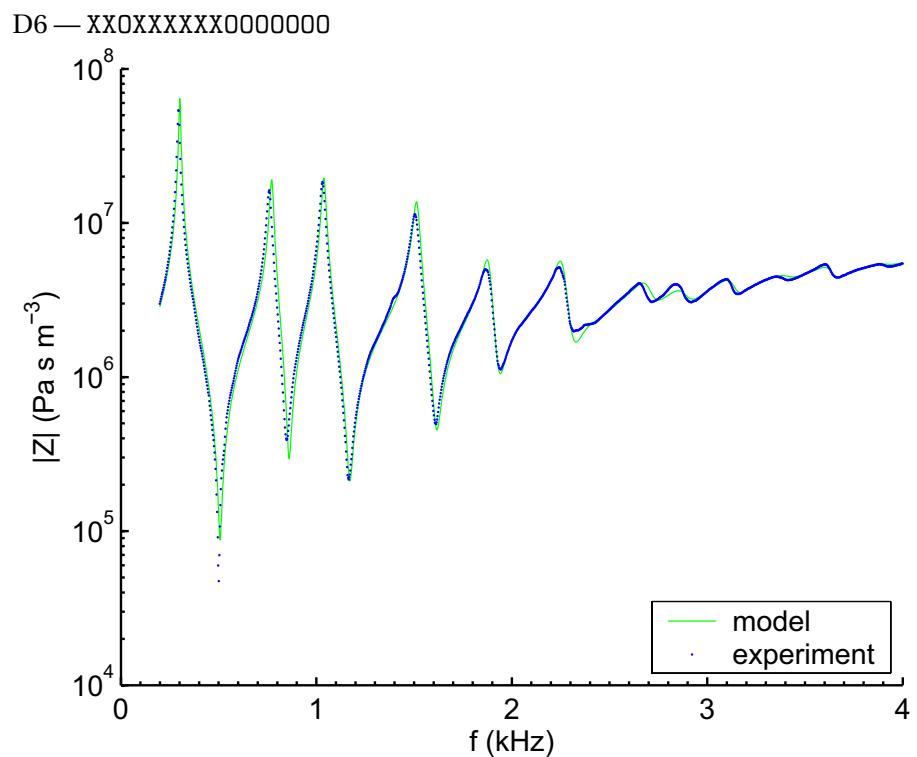
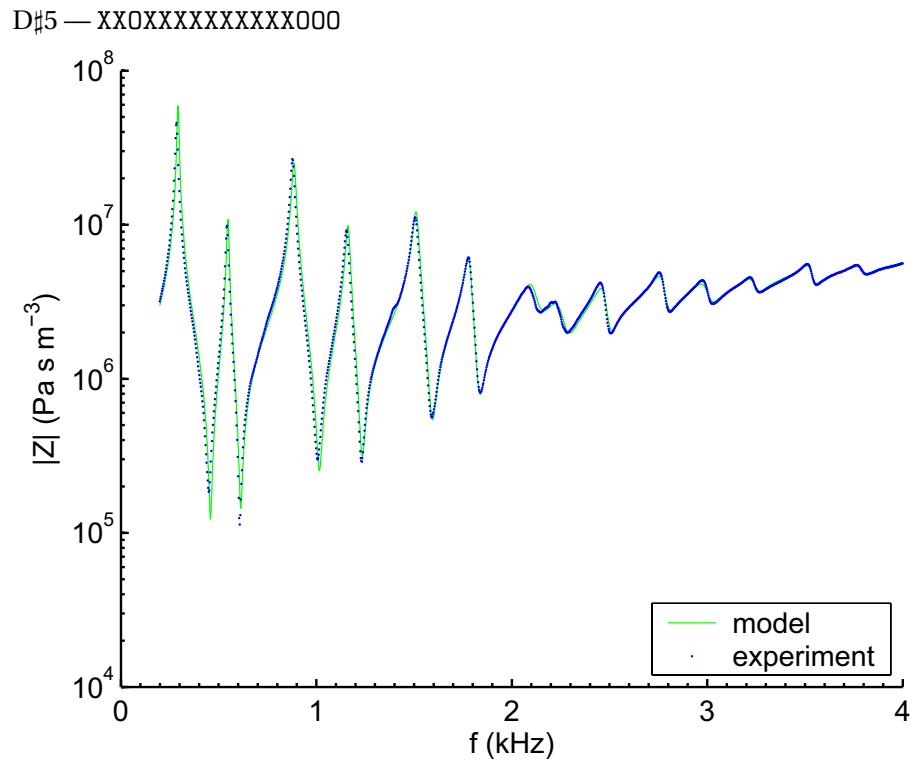




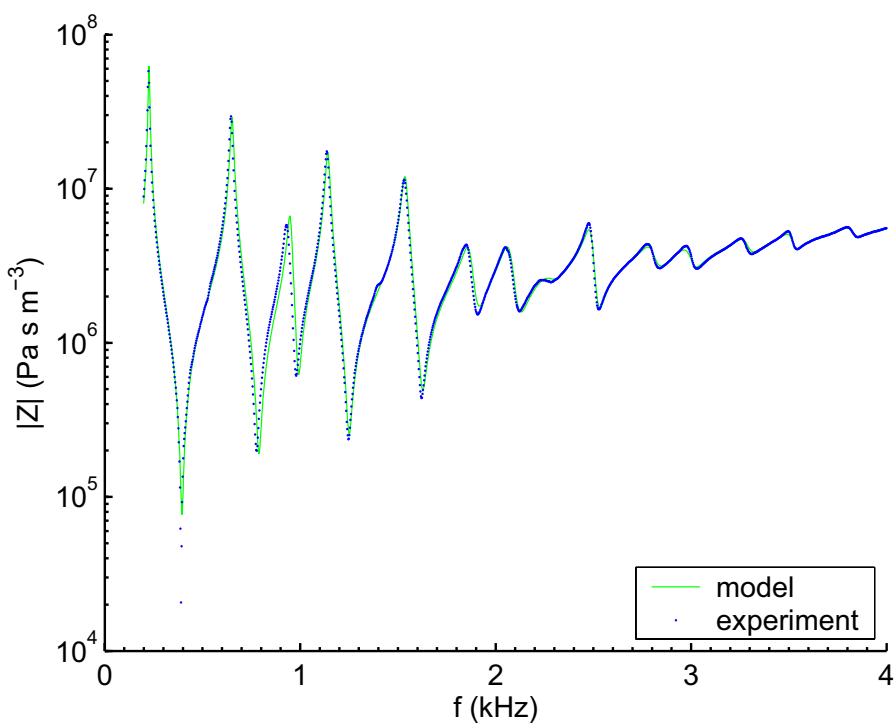
C \sharp 5/6 — XX00000X000000000

D5 — XX0XXXXXXXXXXXXX00

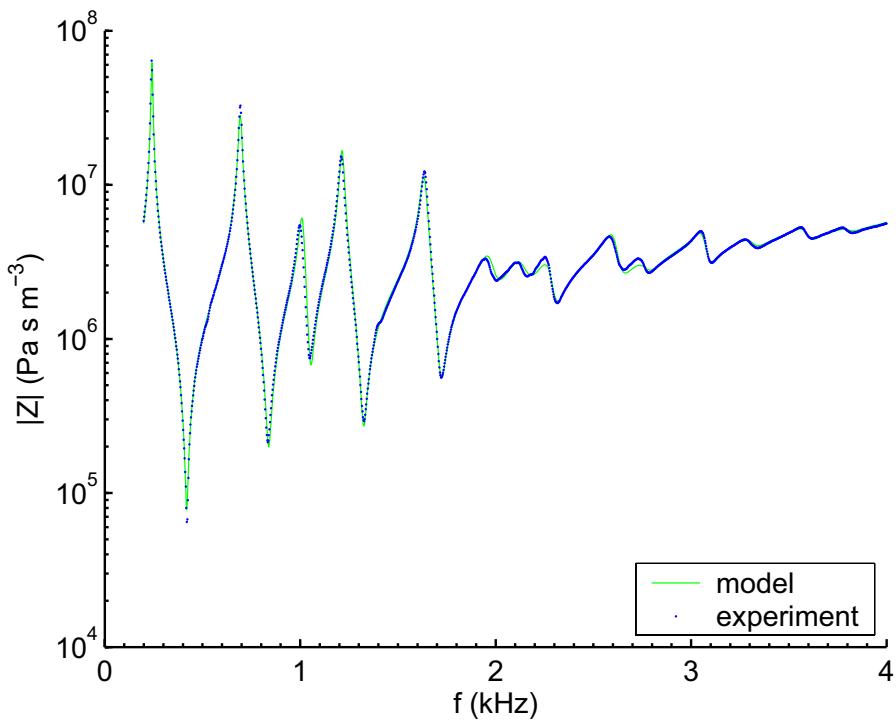


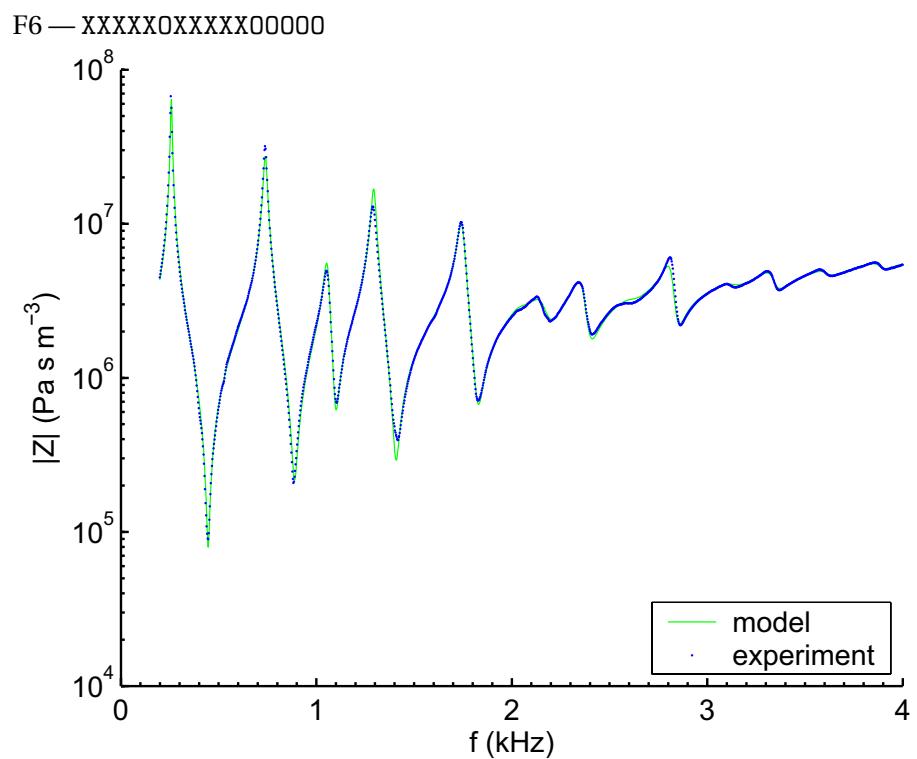
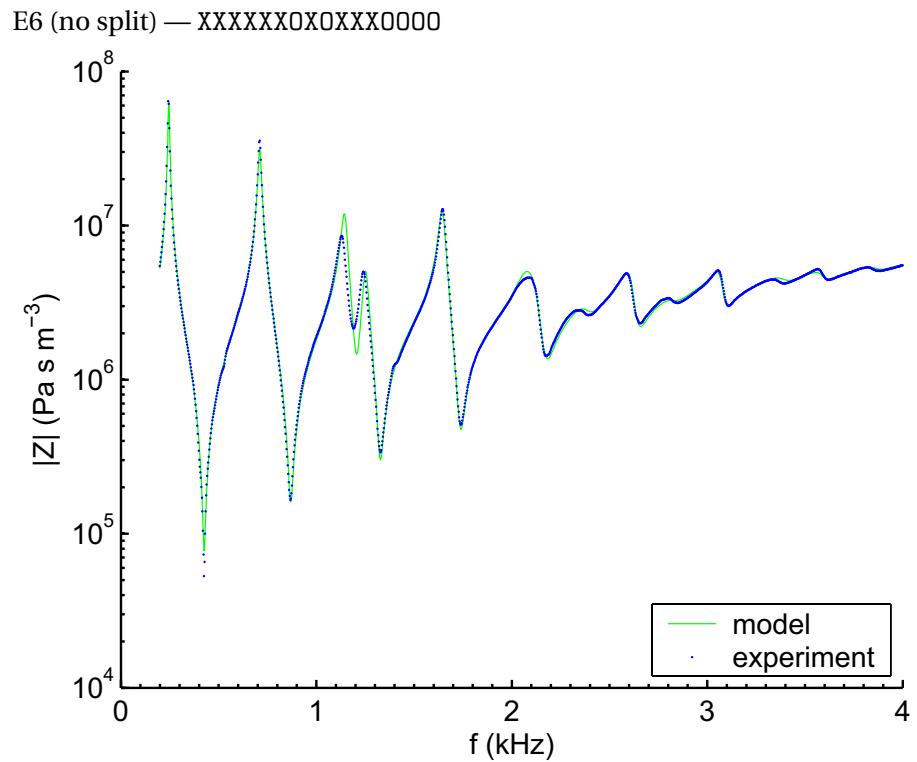


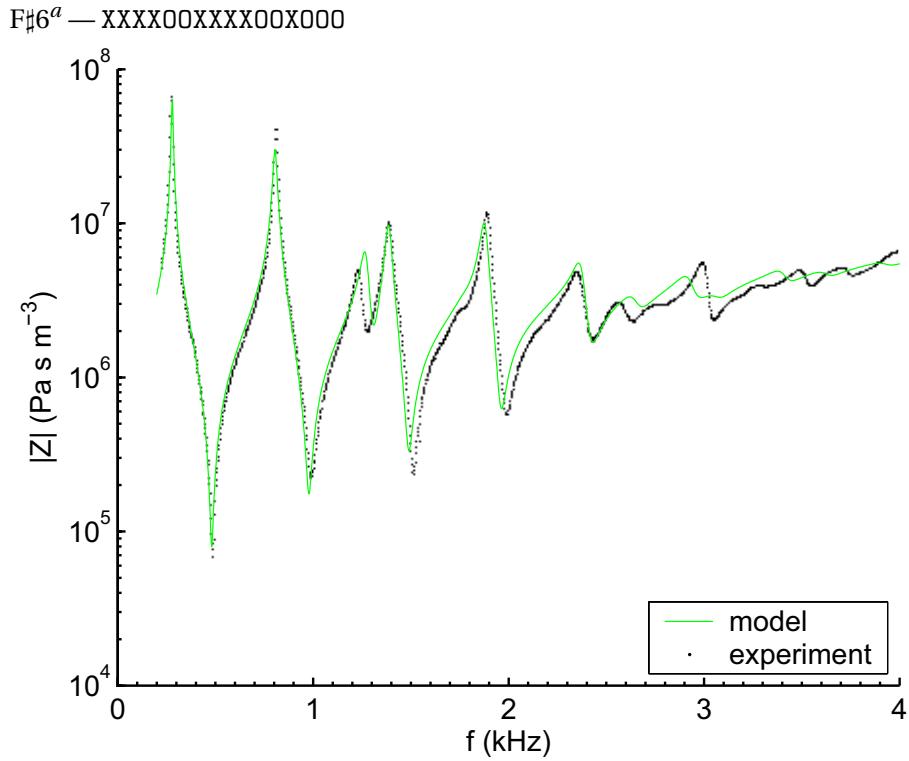
D#6 — XXXXXXXOXXXXX000



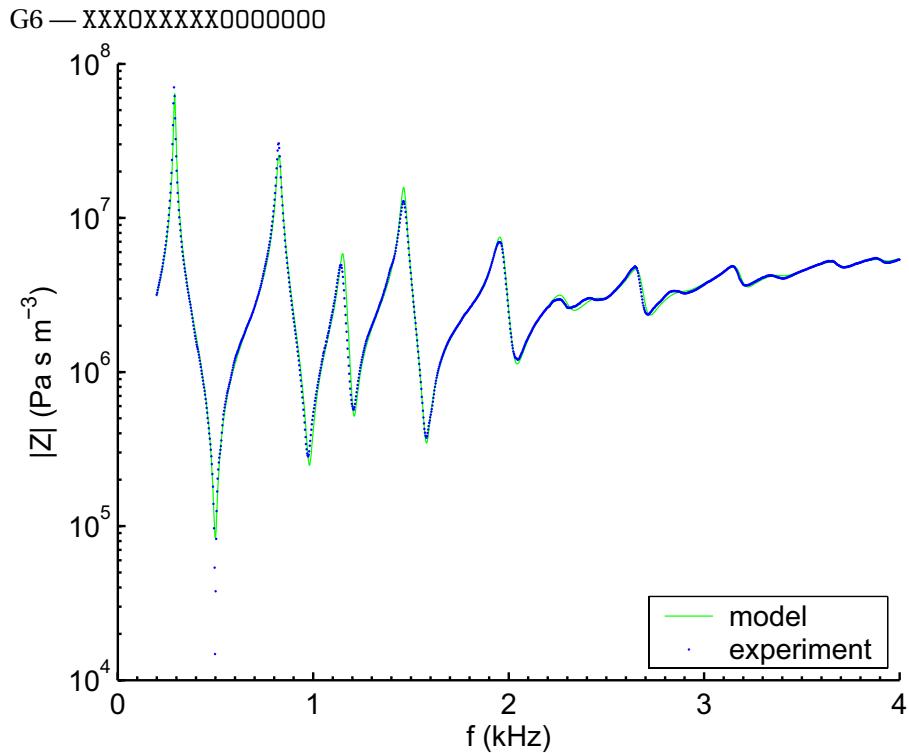
E6 (split) — XXXXXXOXXXXX0000

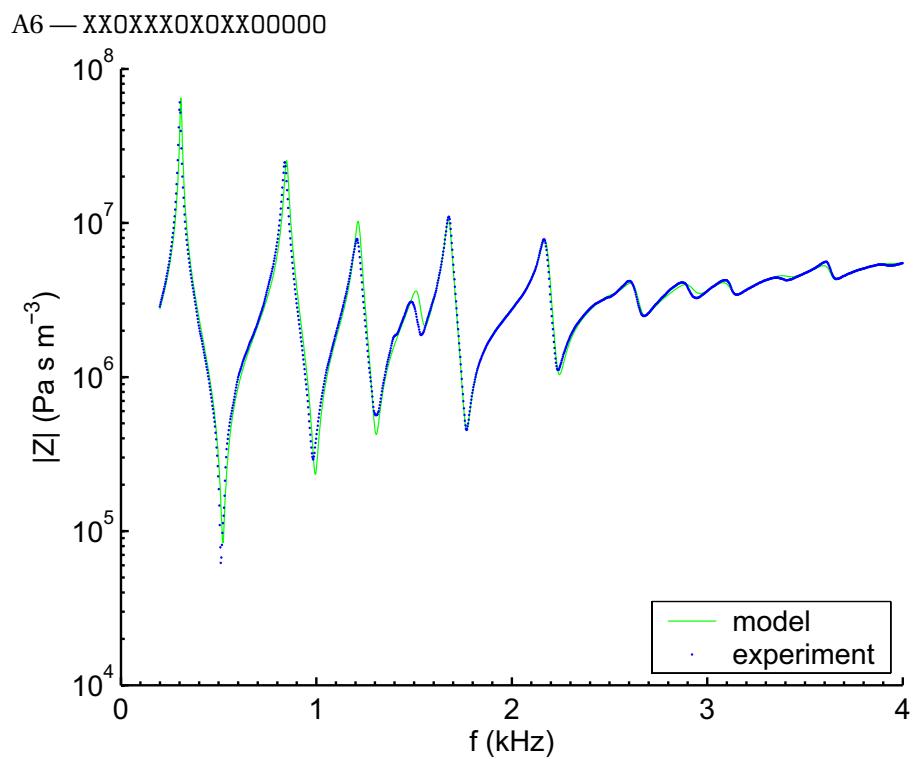
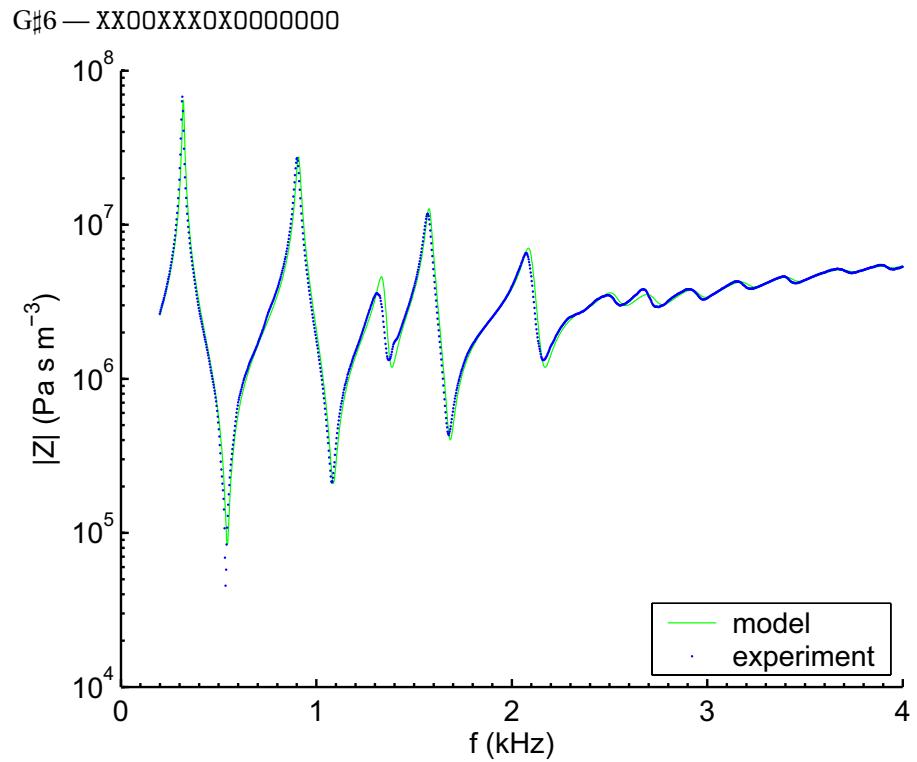




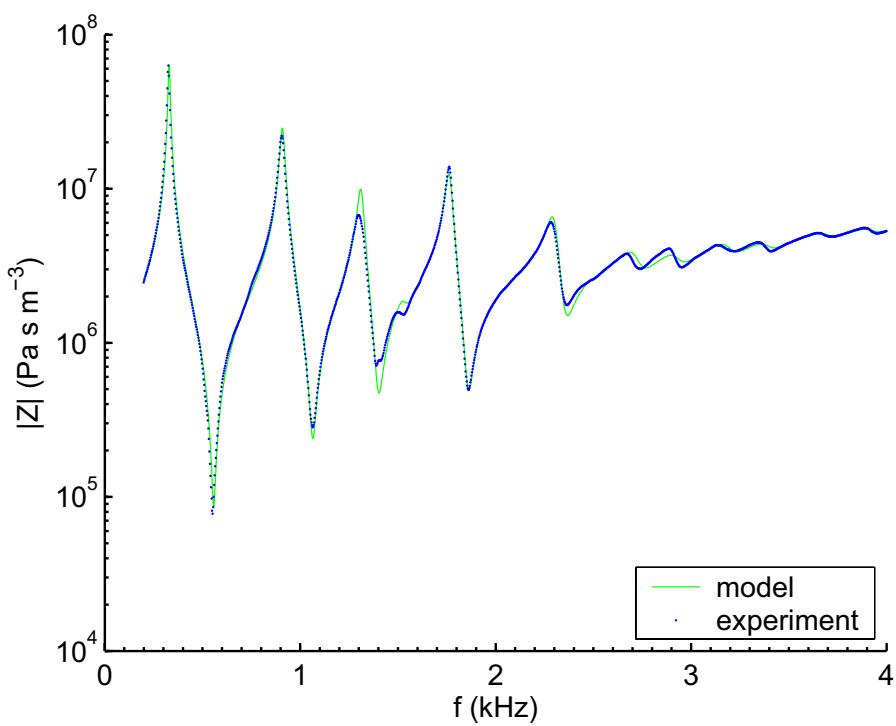


^aThe flute was fingered incorrectly when this spectrum was measured—the experimental data from Wolfe et al. (2001*b*) have been used instead.

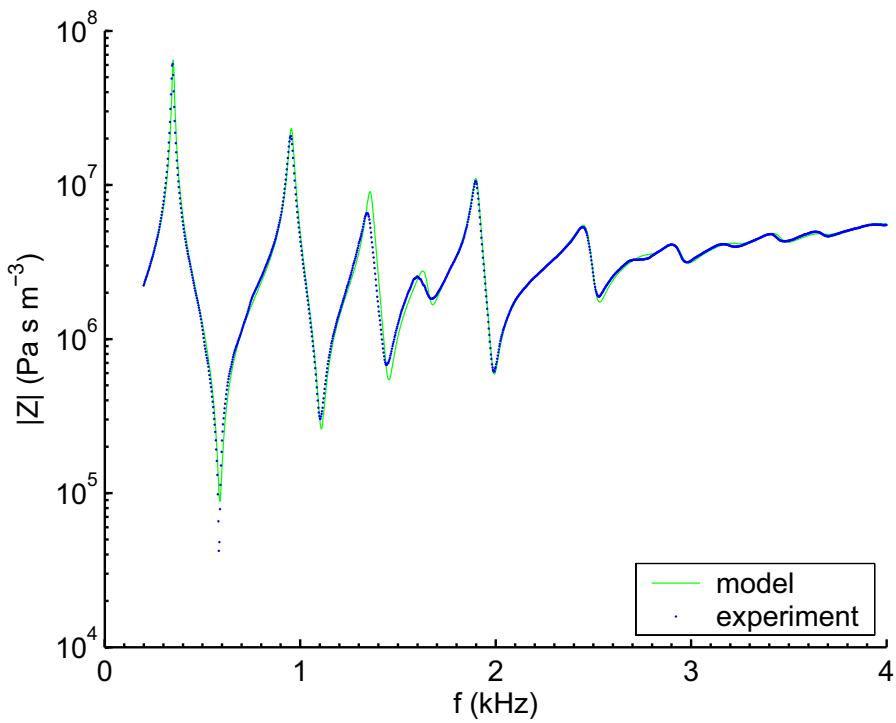


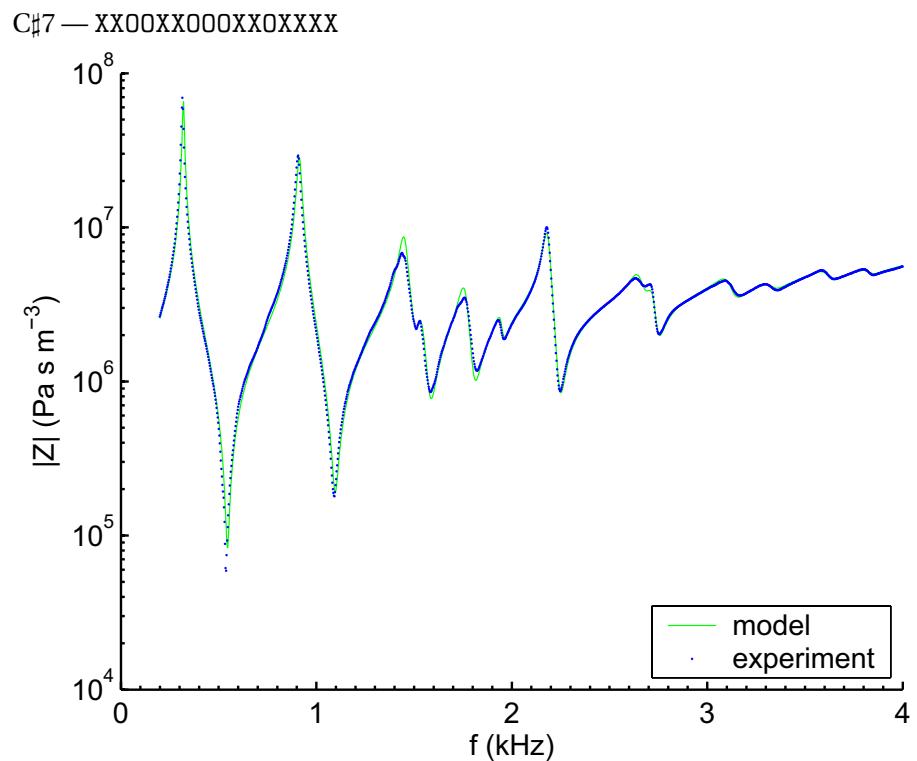
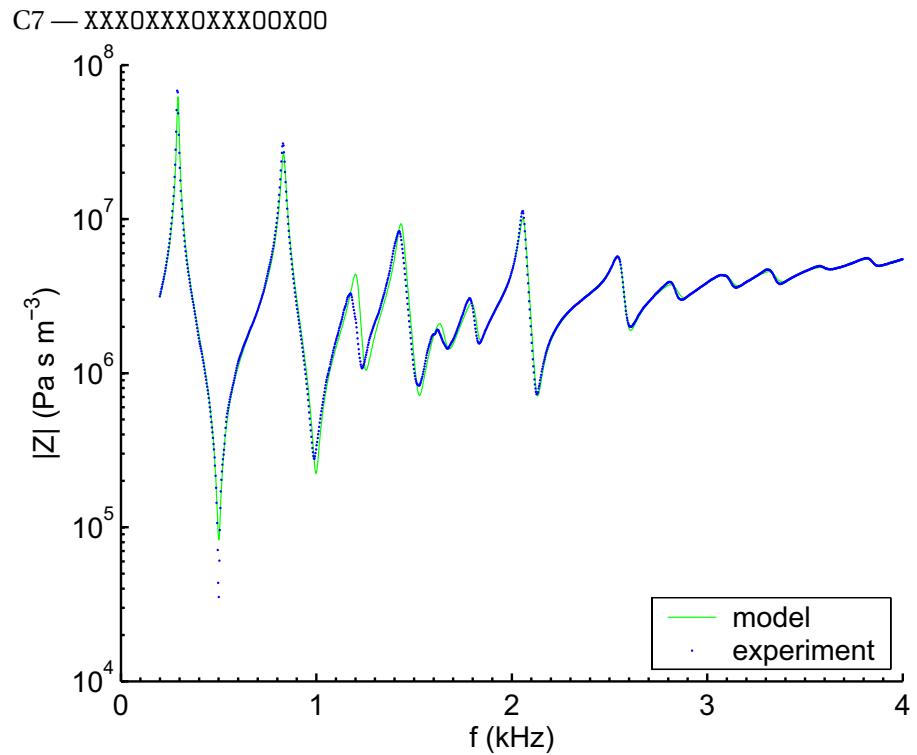


A#6 — X0XXX00X0XX000000

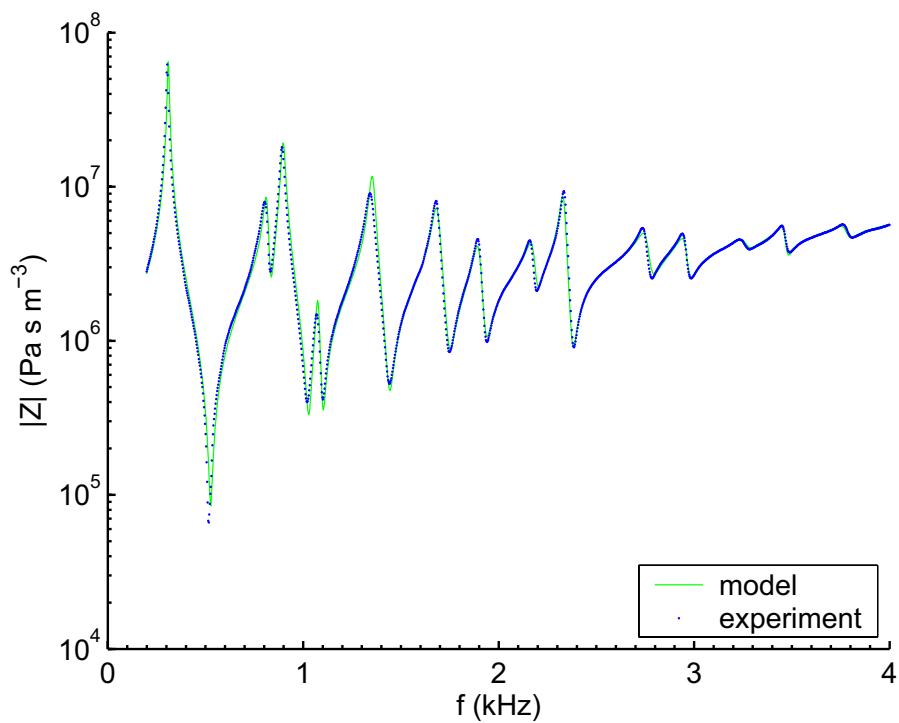
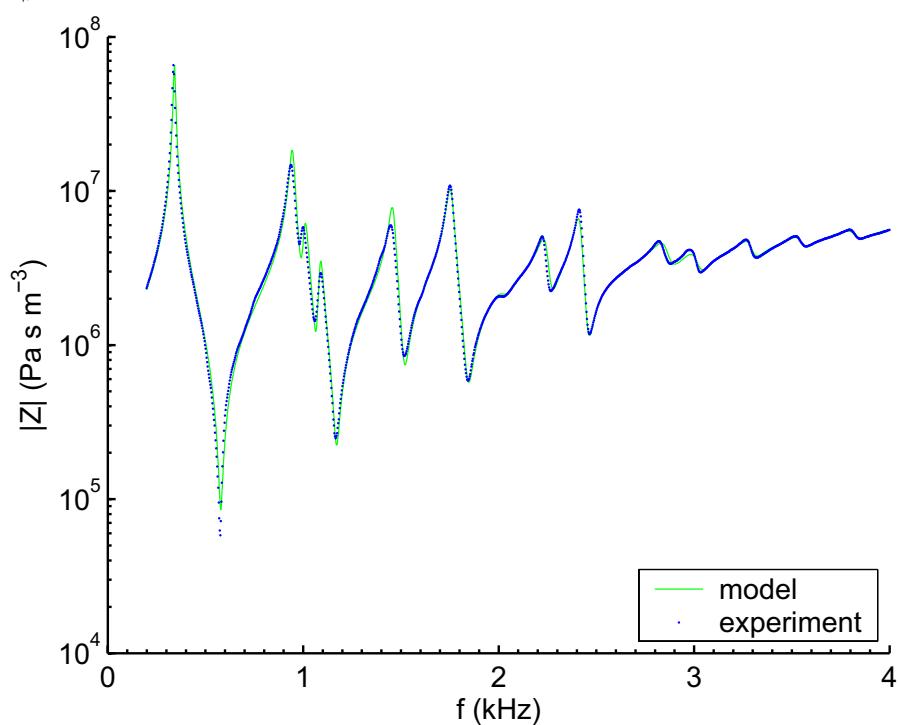


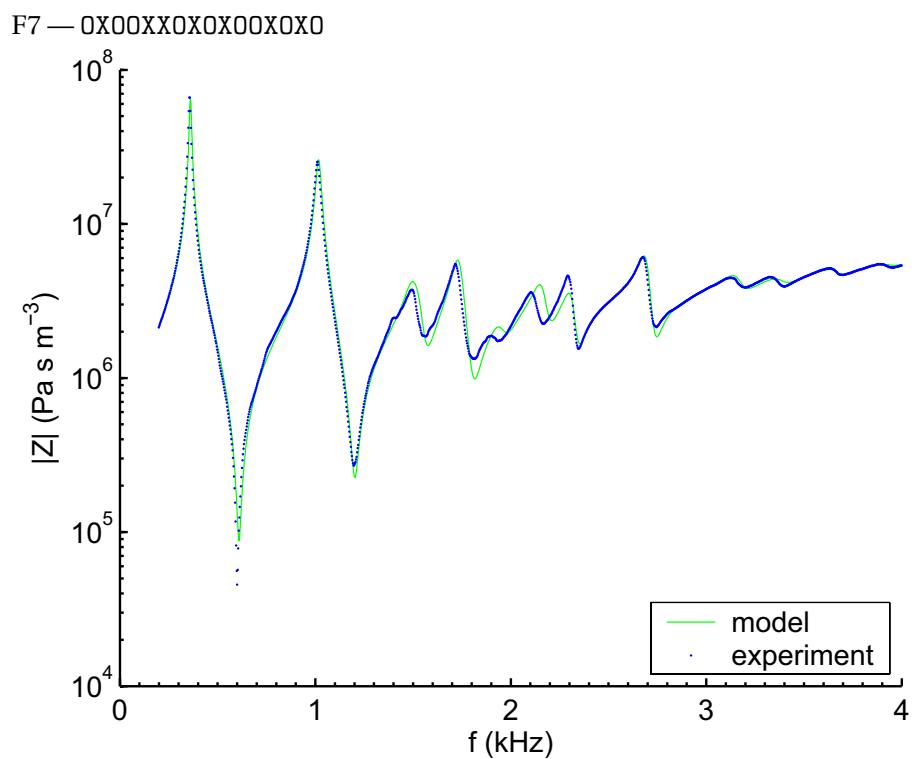
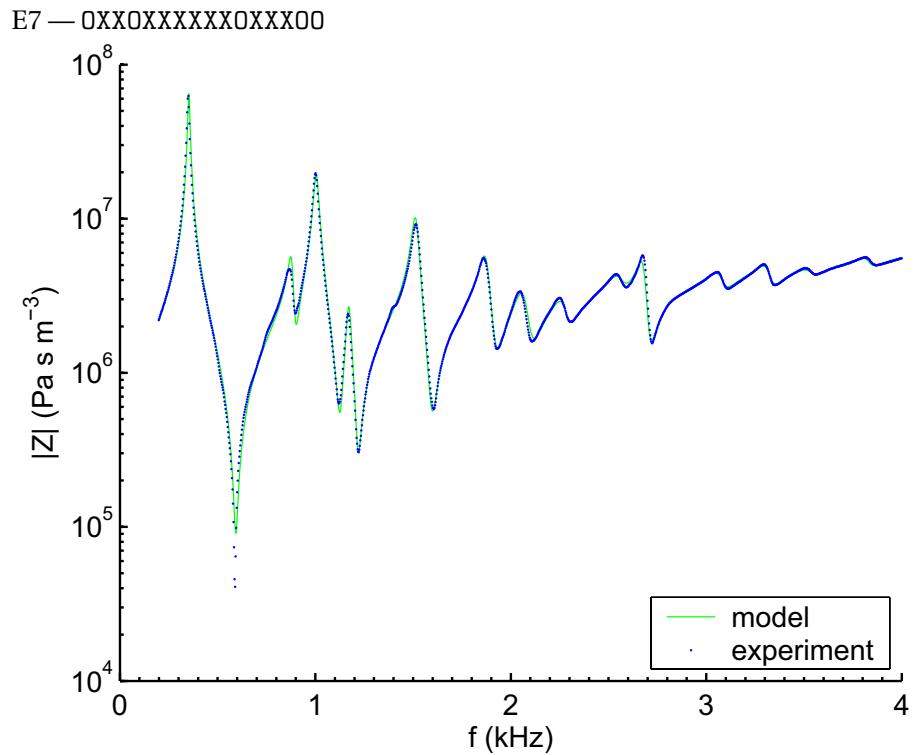
B6 — OXXX00XXX00000000



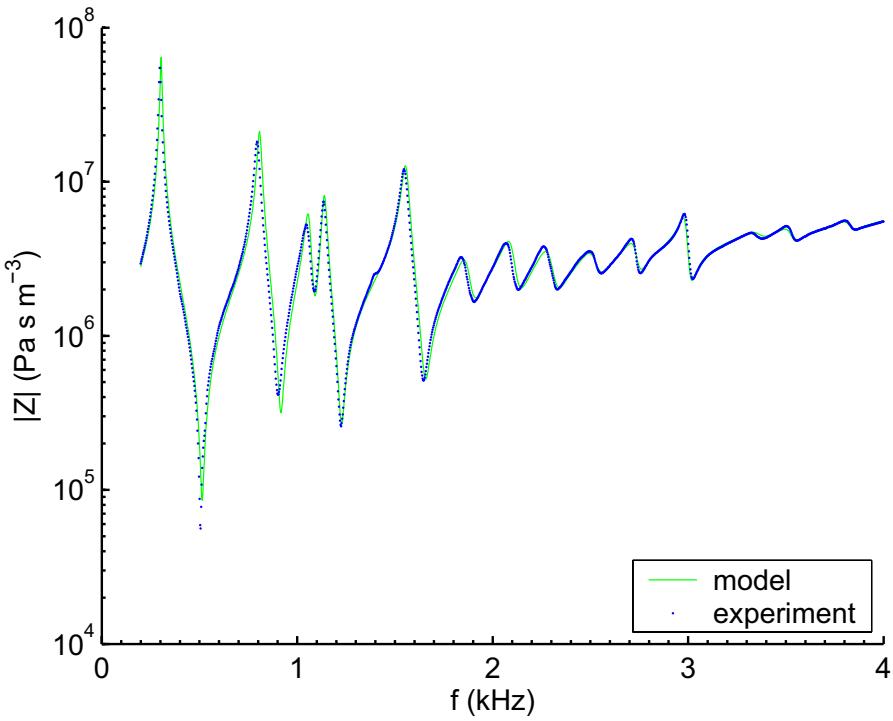


D7 — XXOXXOXXXXXOXXX

D~~#~~7 — XOOXXOXXXXXO0XXX

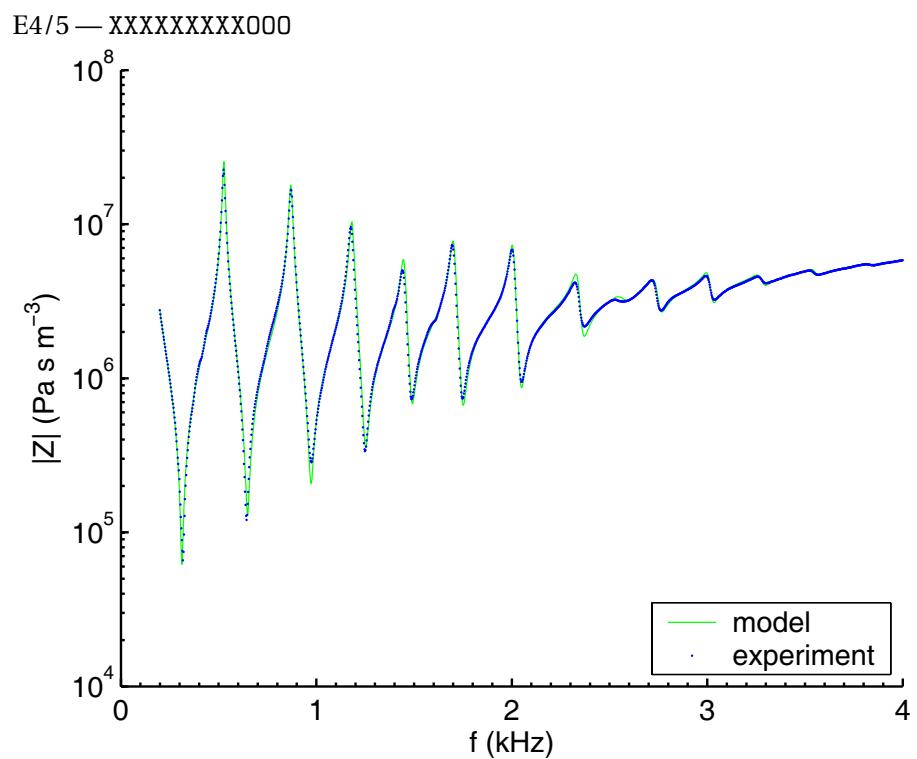
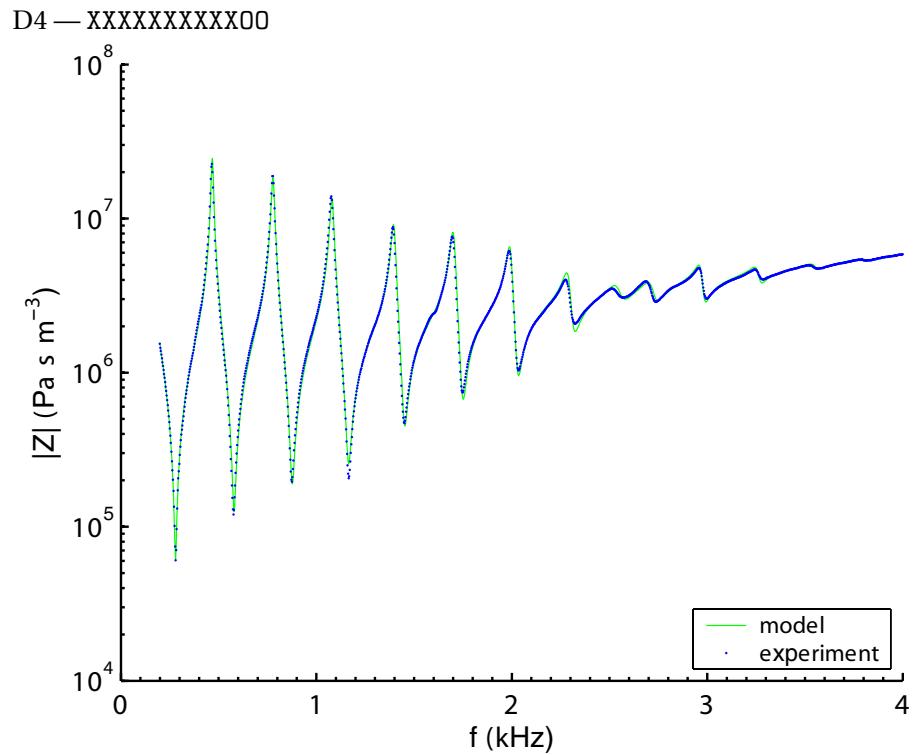


F#7 — XXOXXXXOXXOXXXOO

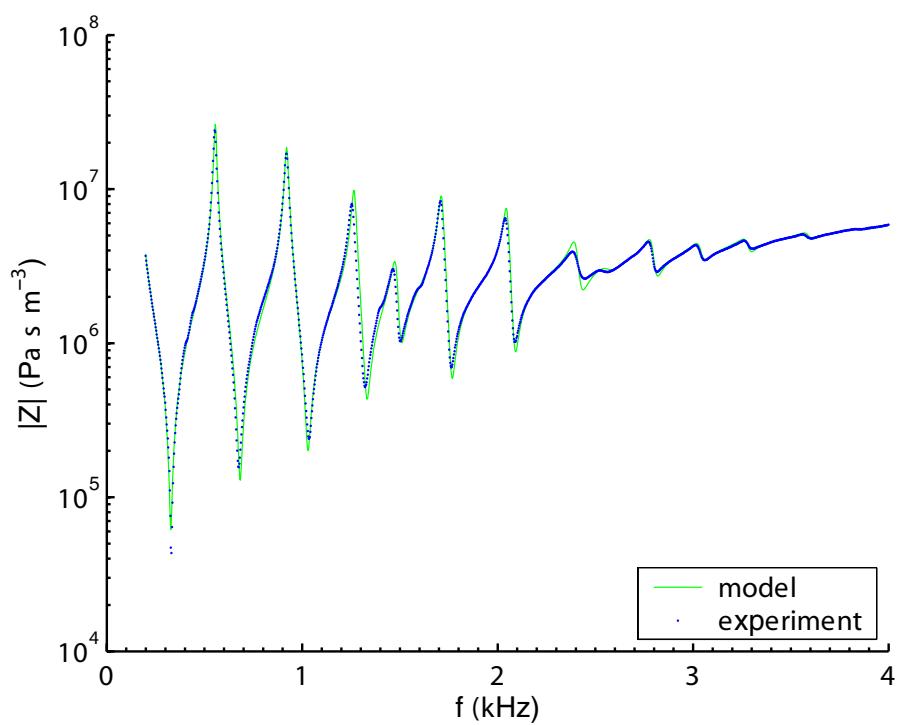


A.2 MCGEE CLASSICAL FLUTE

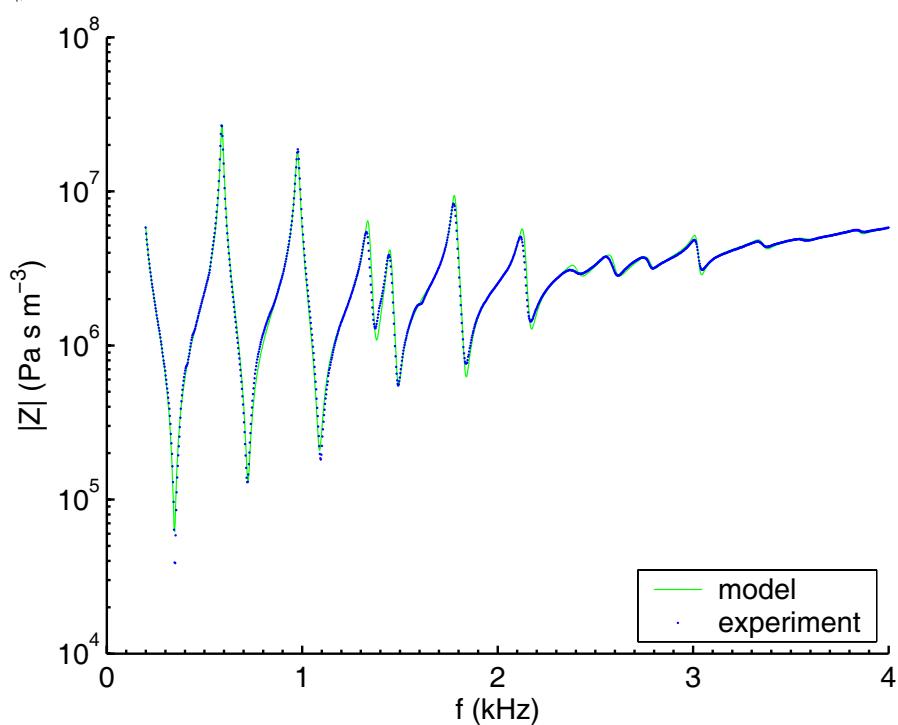
The following graphs show the magnitude of the measured input impedance spectra for a classical flute (made by Terry McGee, Canberra) for all recommended fingerings. The tuning slide was set at 12.8 mm and the cork at 19.0 mm. The impedance spectra were measured using an impedance head of diameter 7.8 mm. The inertance of a stub of air of length 5 mm and diameter 7.8 mm was added to the measurements and simulations in order to approximate the effect of a player's face impedance and to facilitate comparison with earlier measurements (Wolfe et al. 2001a). The measurements were made at $T = 21.6 \pm 0.5^\circ\text{C}$ and relative humidity $37 \pm 1\%$.

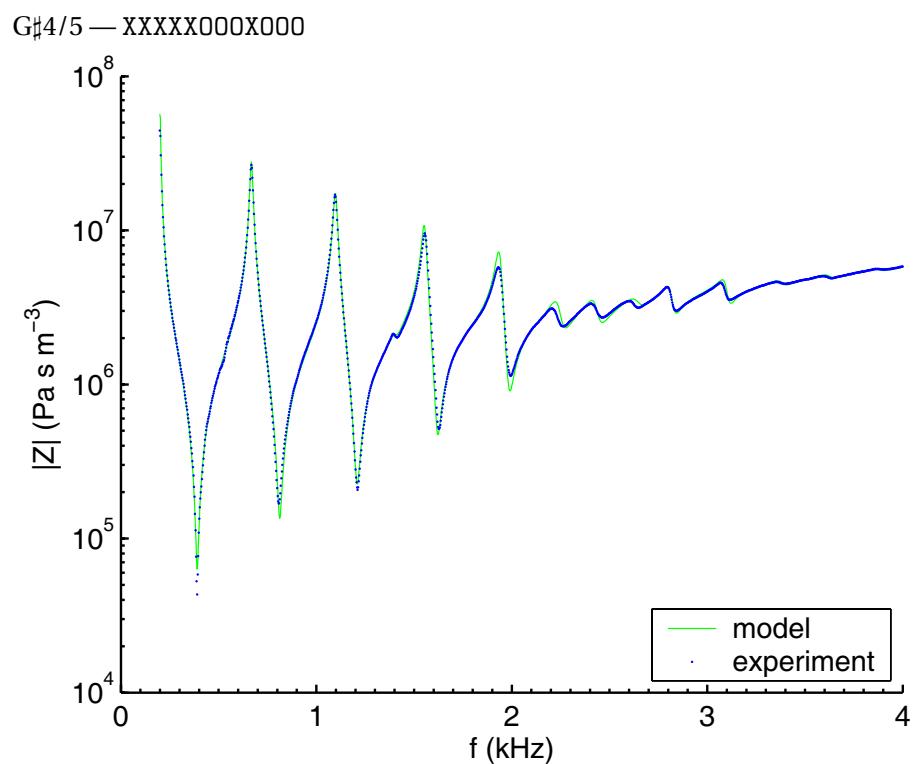
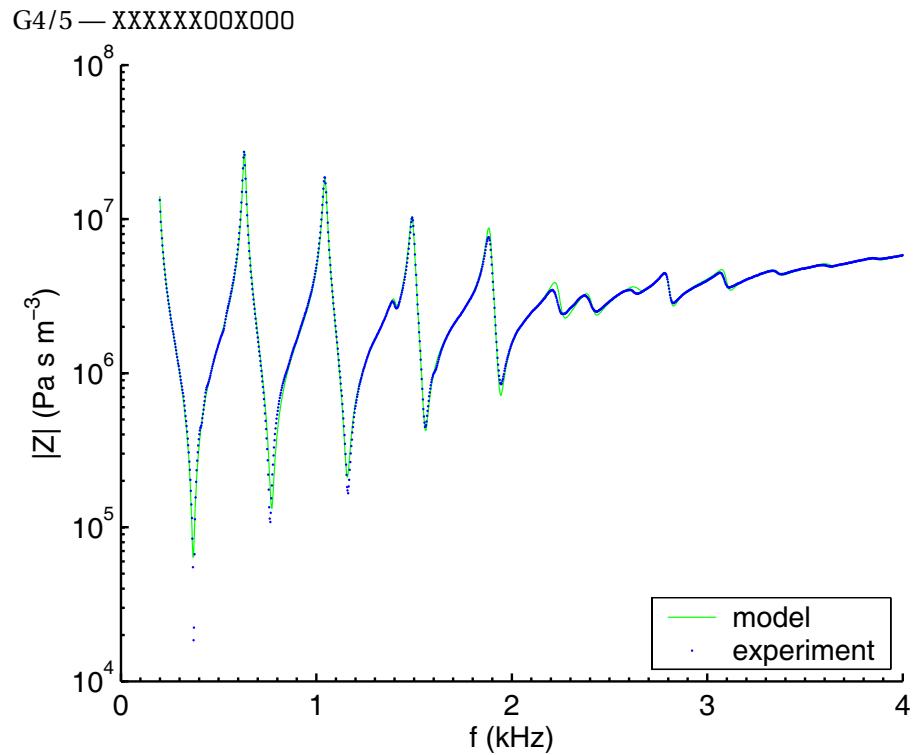


F4/5 — XXXXXXXX0000

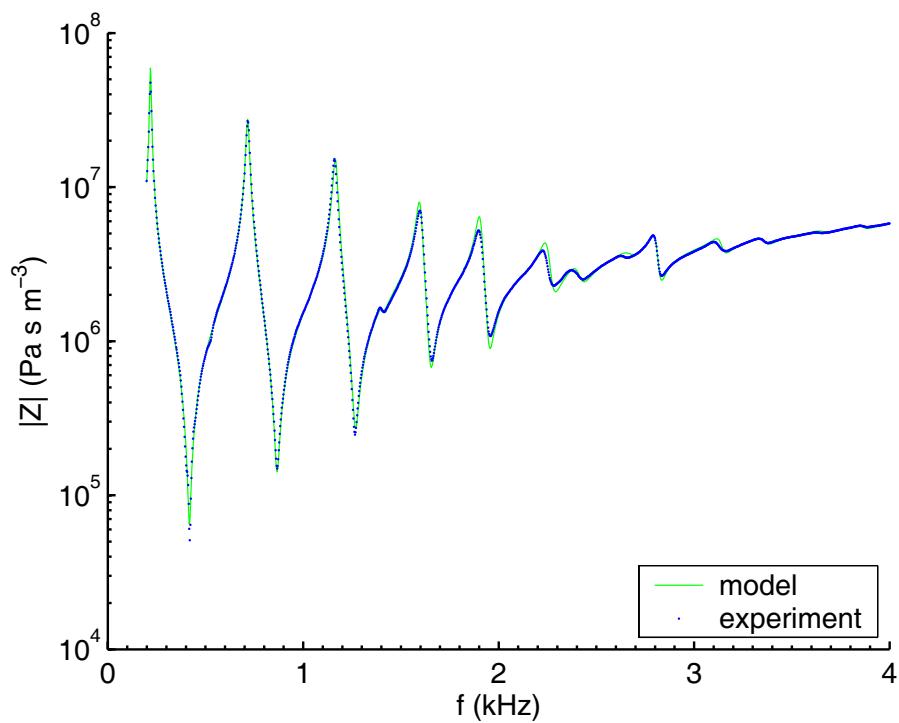


F#4/5 — XXXXXXXXOX000

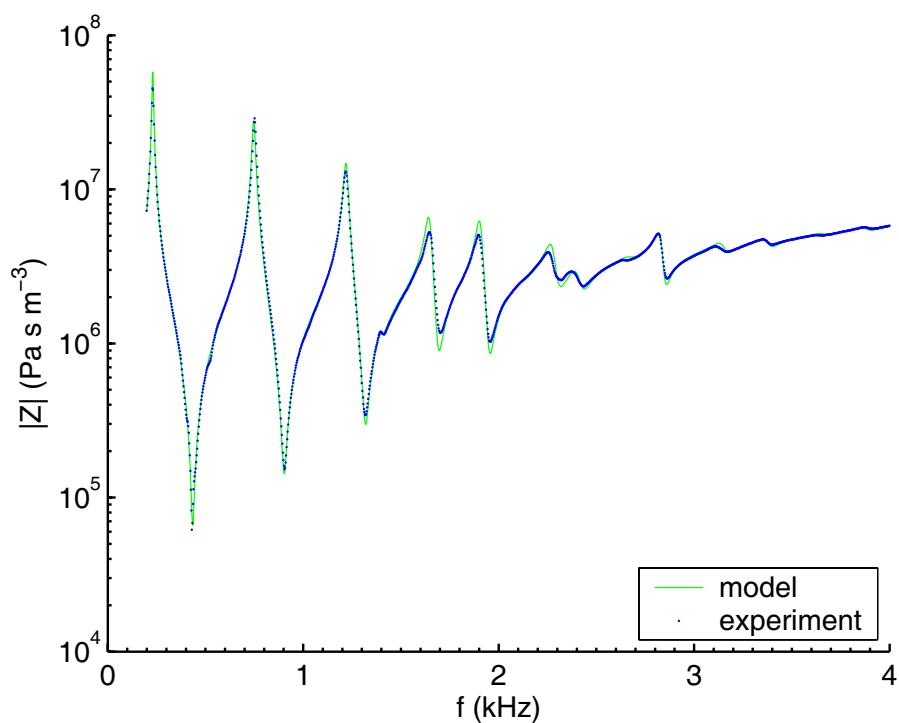


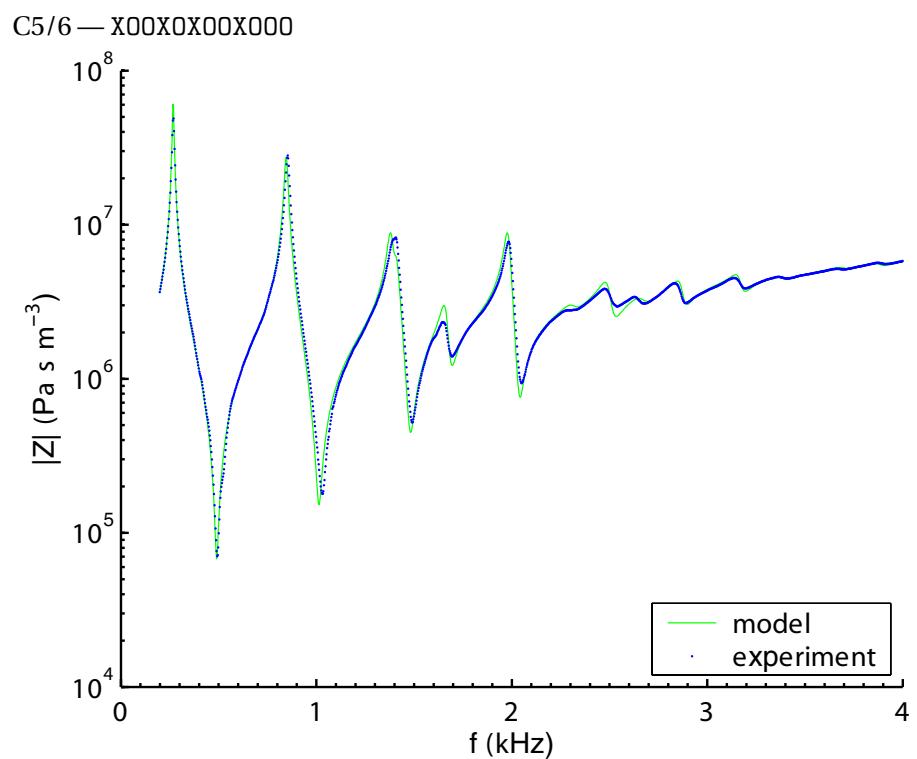
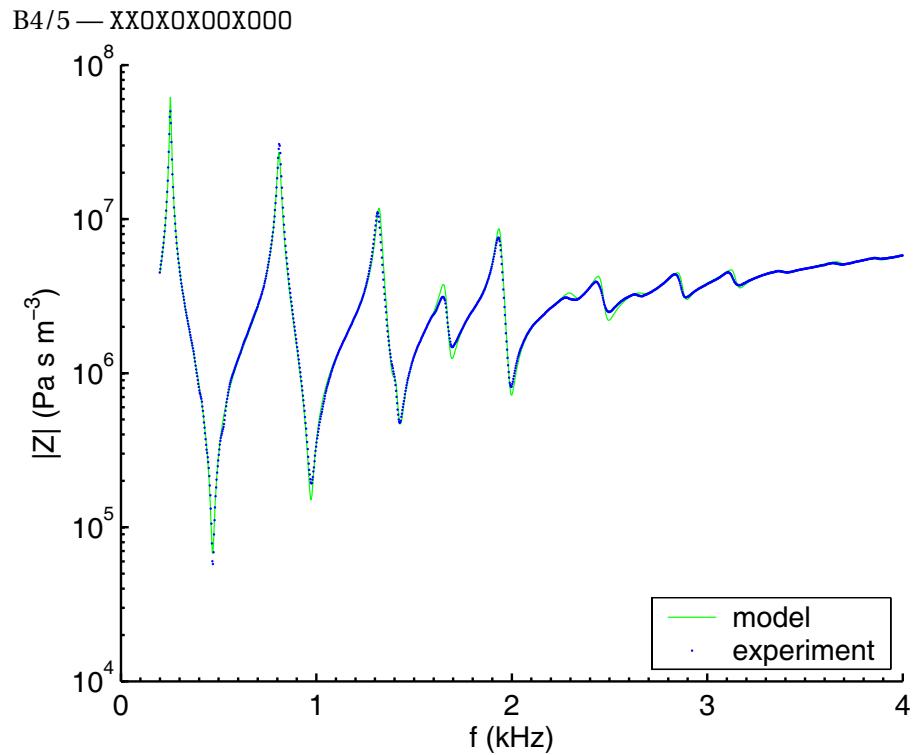


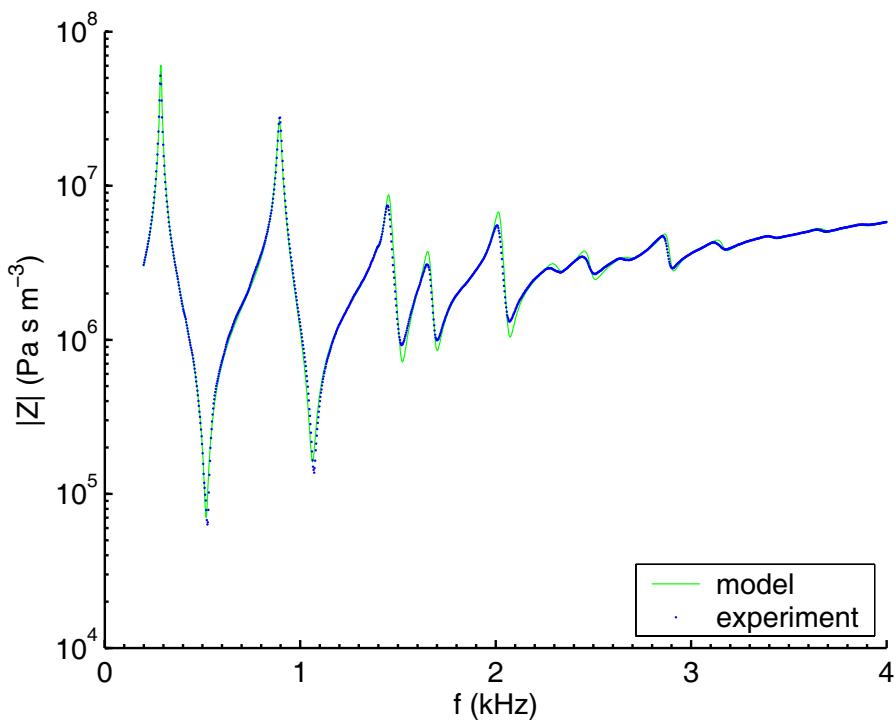
A4/5 — XXXXOXOOX000



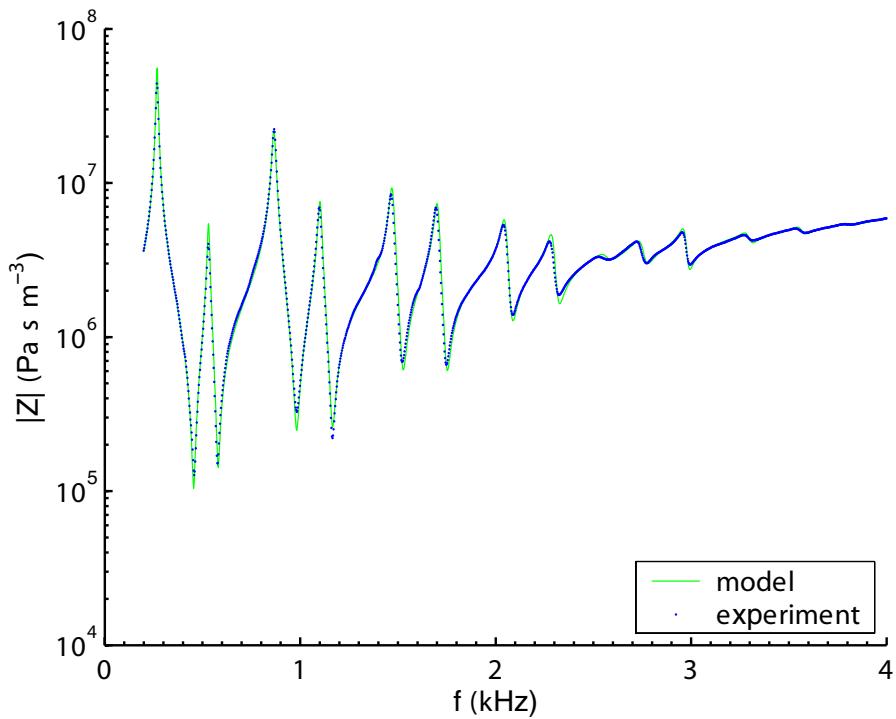
A#4/5 — XXX00X00X000

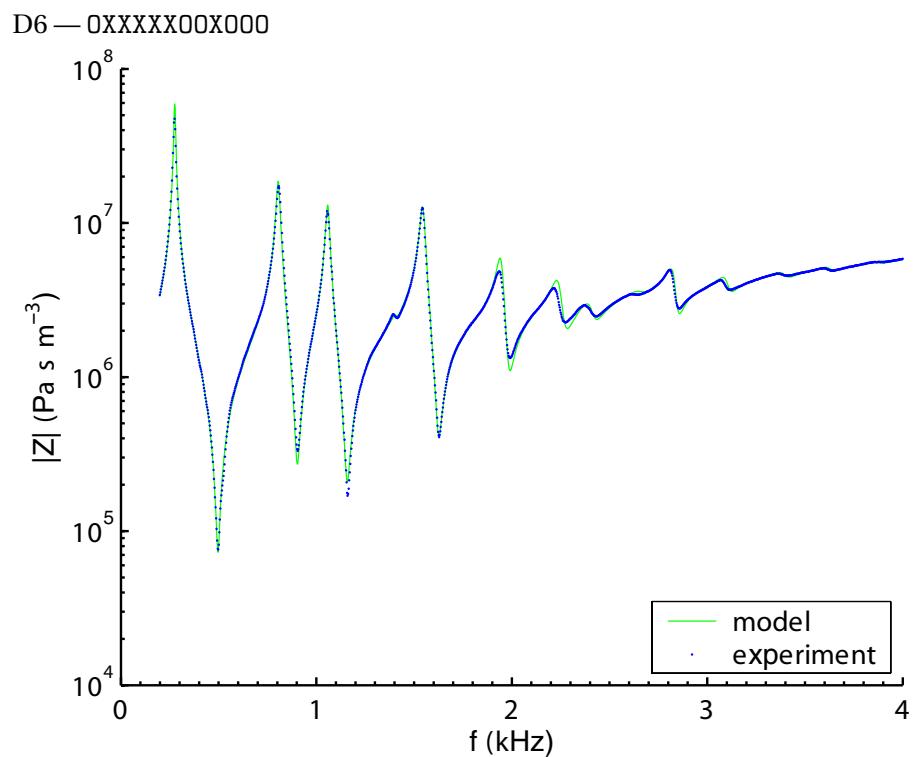
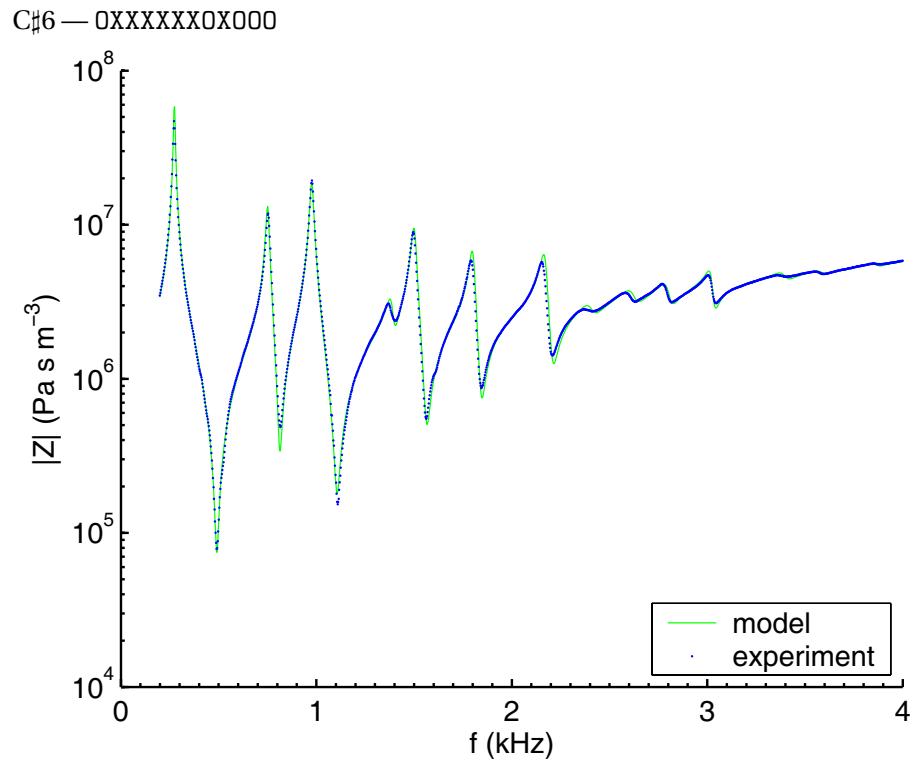


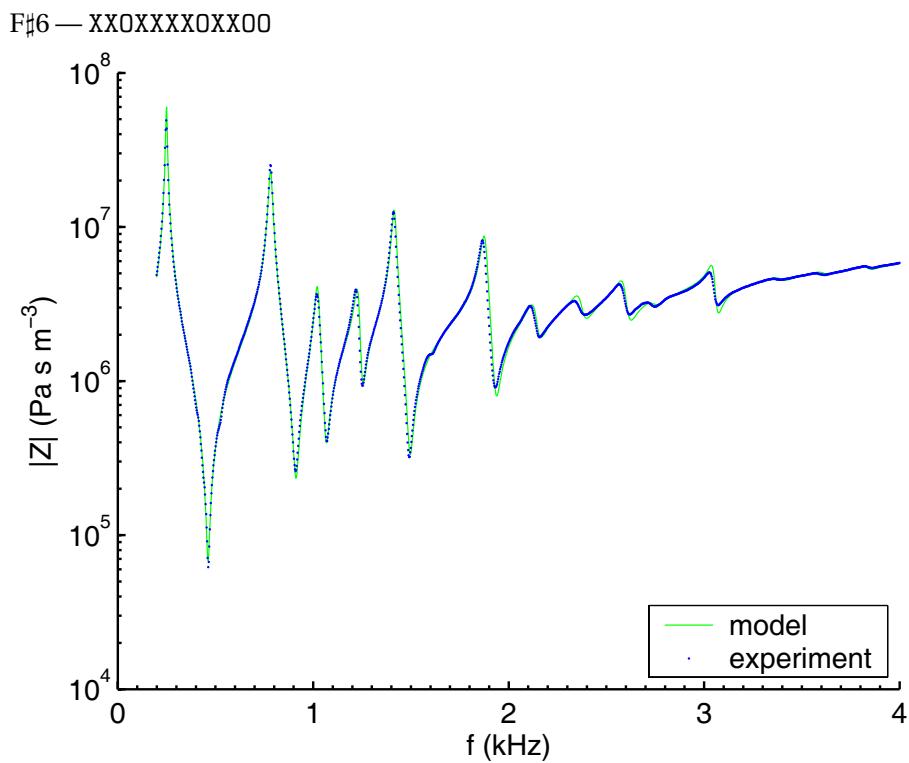
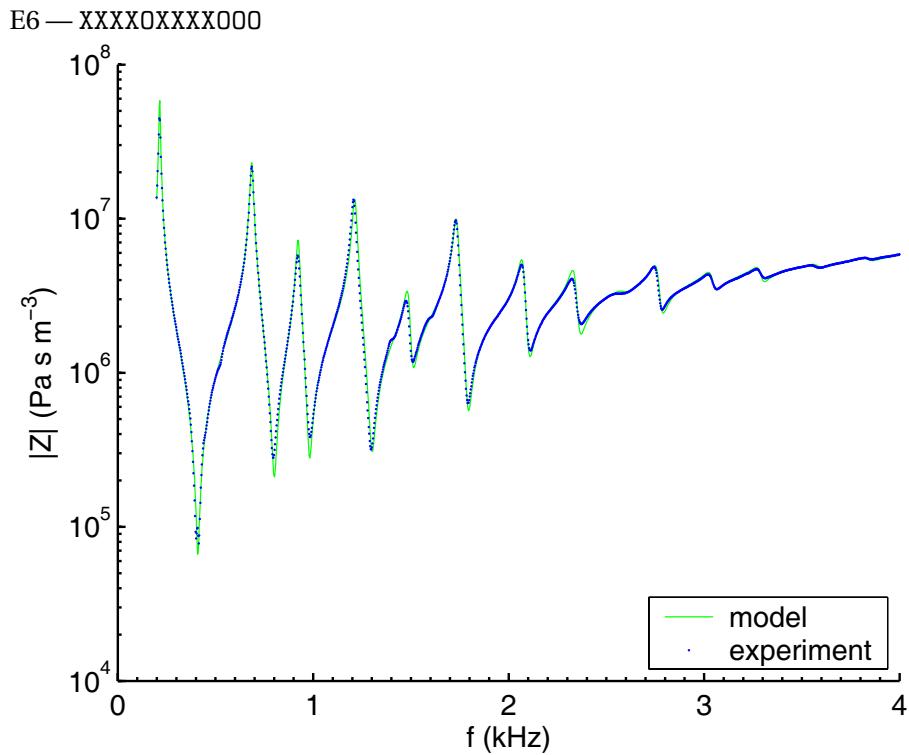


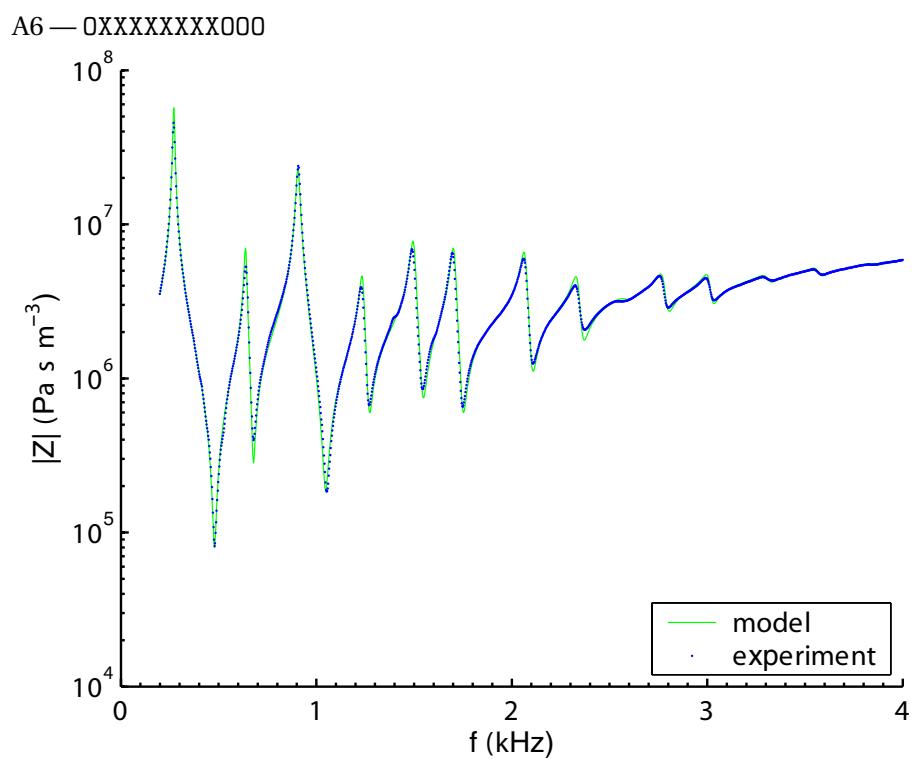
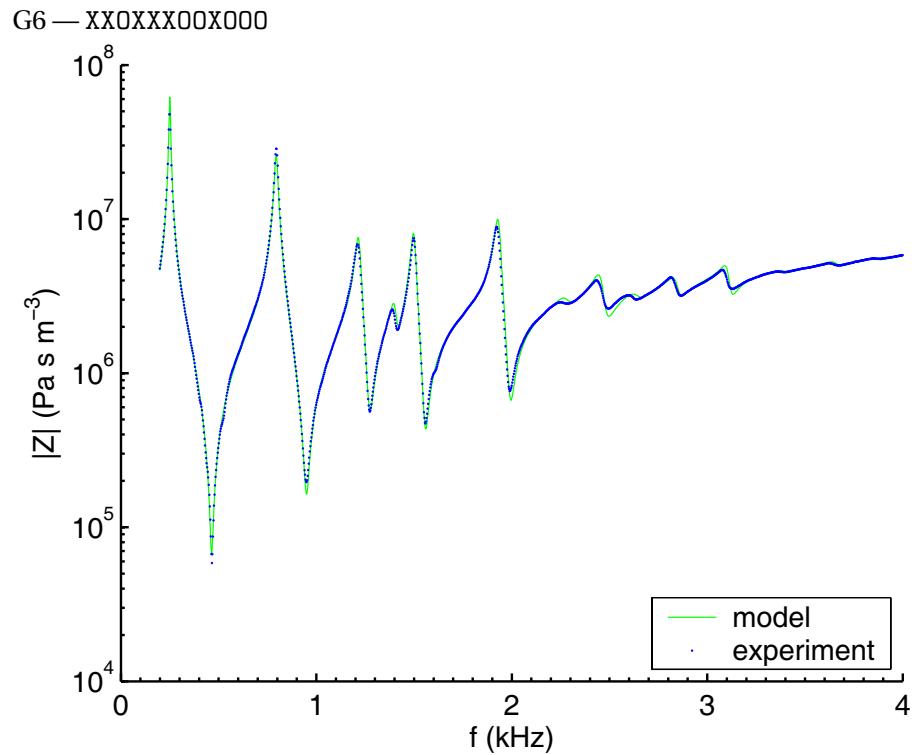
C \sharp 5 — OXOXOXOOXOOO

D5 — OXXXXXXXXXXOO





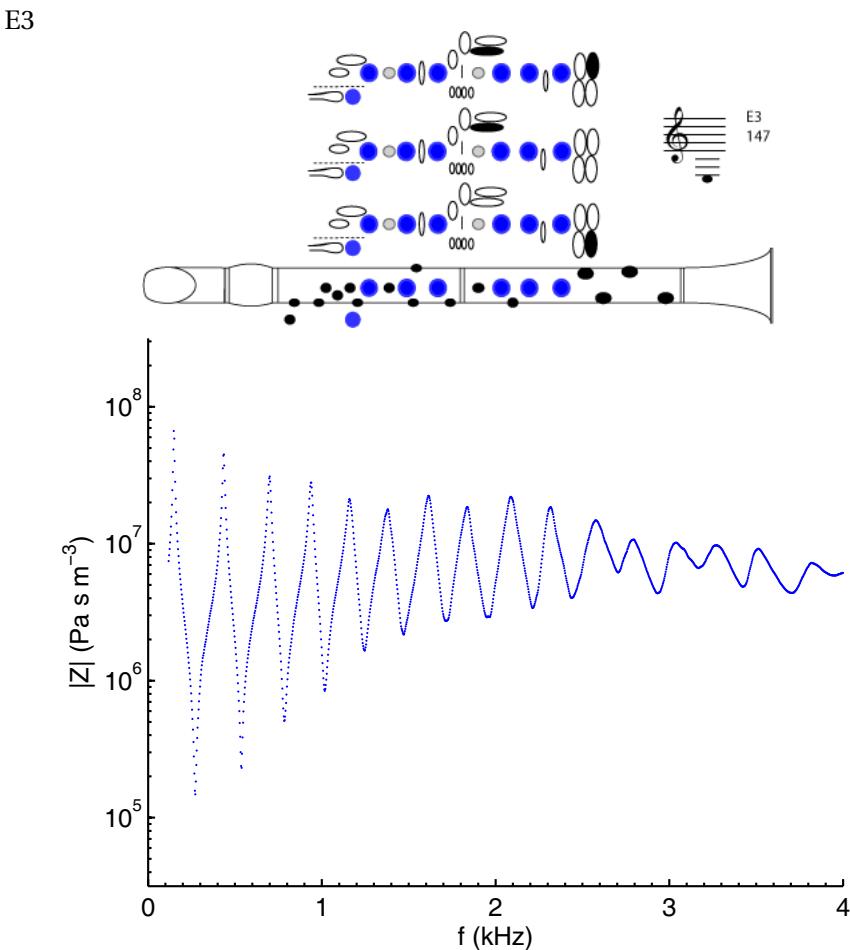


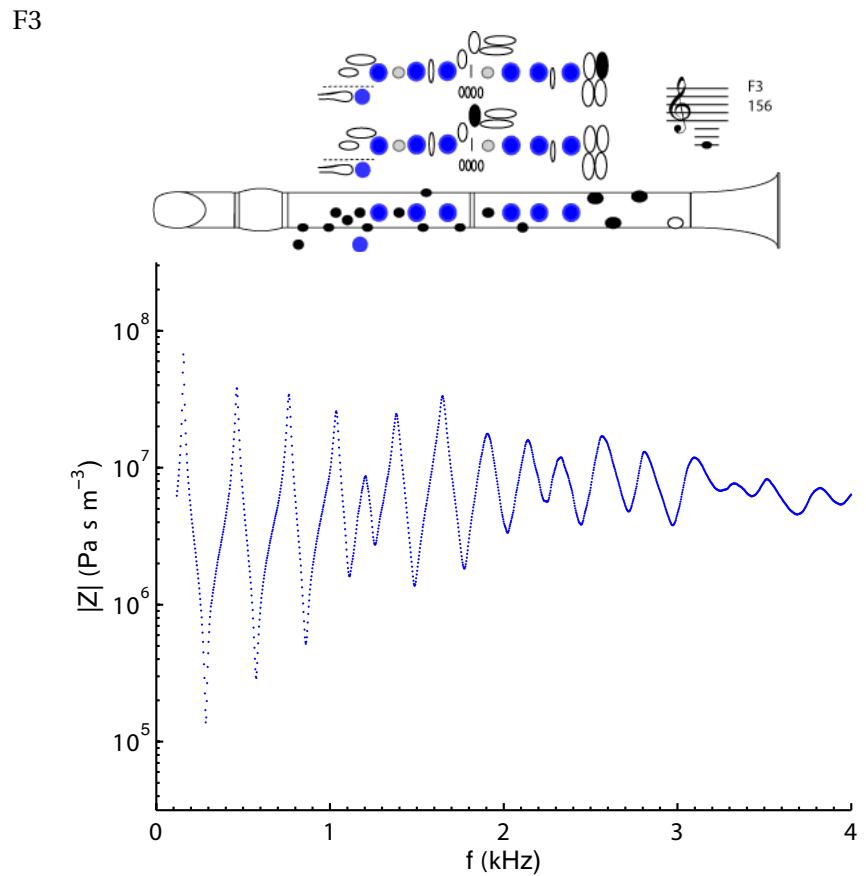


A.3 YAMAHA B \flat CLARINET

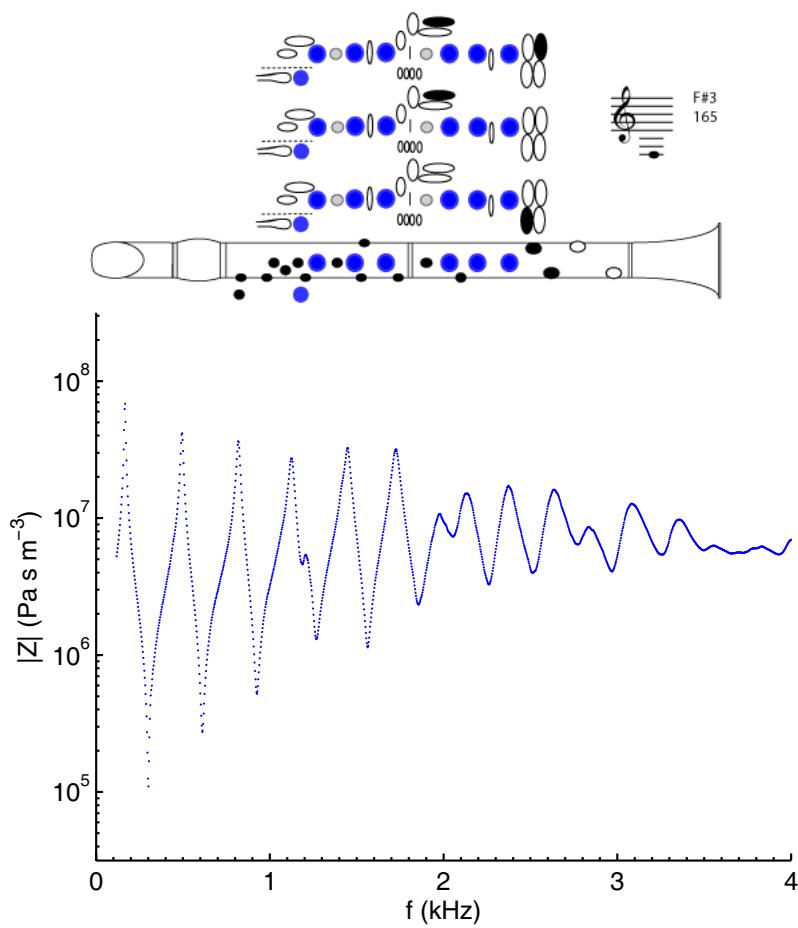
The following graphs show the magnitude of the measured input impedance spectra for a B \flat clarinet (Yamaha Custom CX) for all recommended fingerings. The impedance shown has been corrected to include the reed compliance, as discussed in Chapter 5. The measurements were made at $T = 27.5 \pm 0.5$ °C and relative humidity 61 ± 1 %.

Measurements were also made on an A clarinet and for several alternate and mutiphonic fingerings. For brevity these are not reproduced here. For a complete set of measurements together with sound files and comments for each note see <<http://www.phys.unsw.edu.au/music/clarinet/>>.

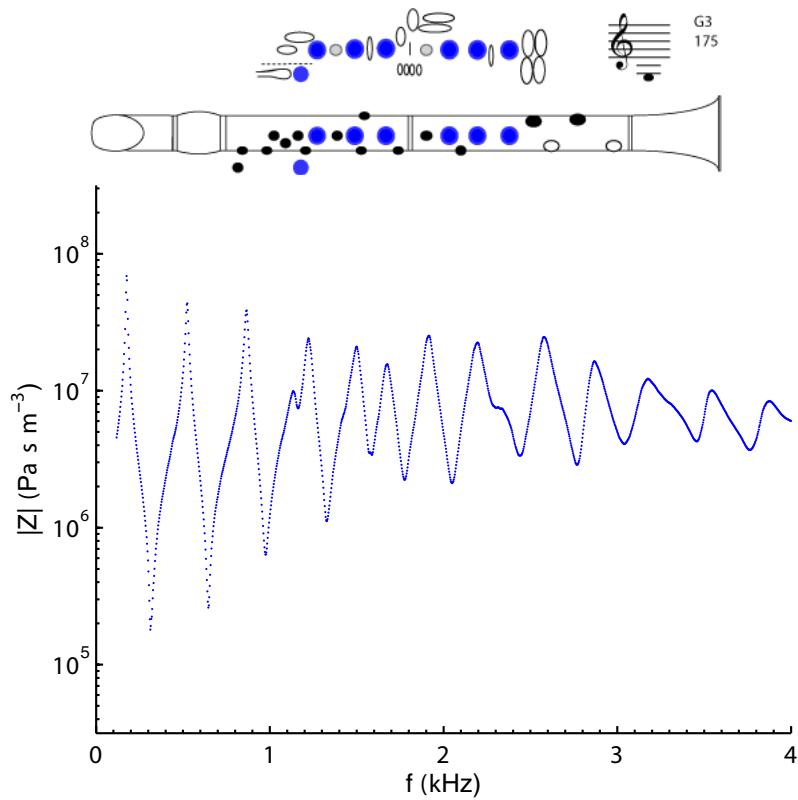


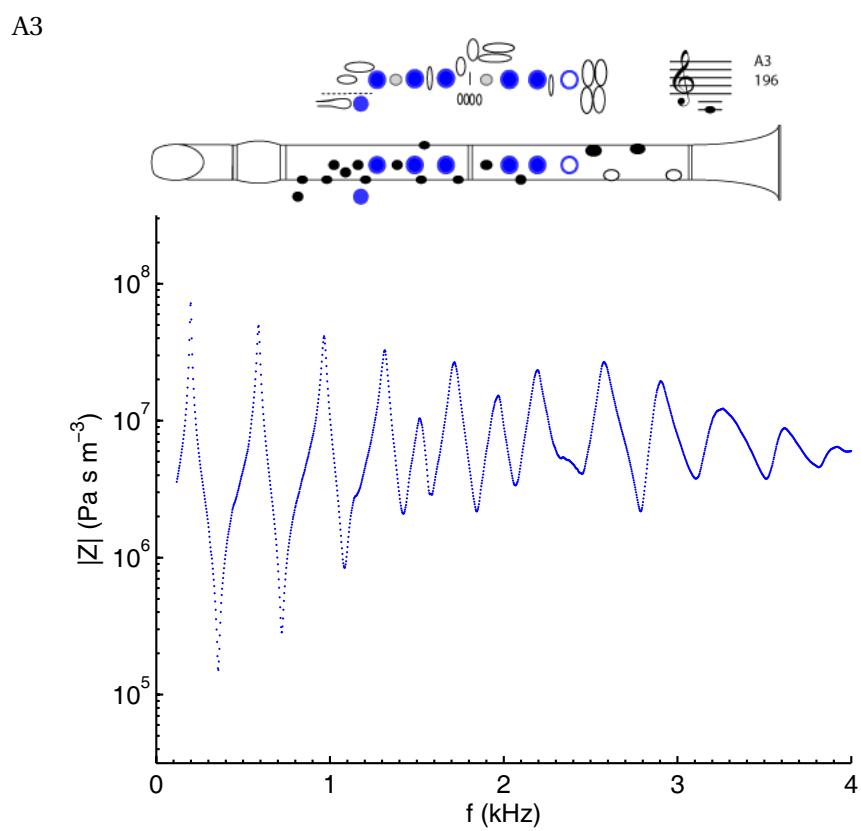
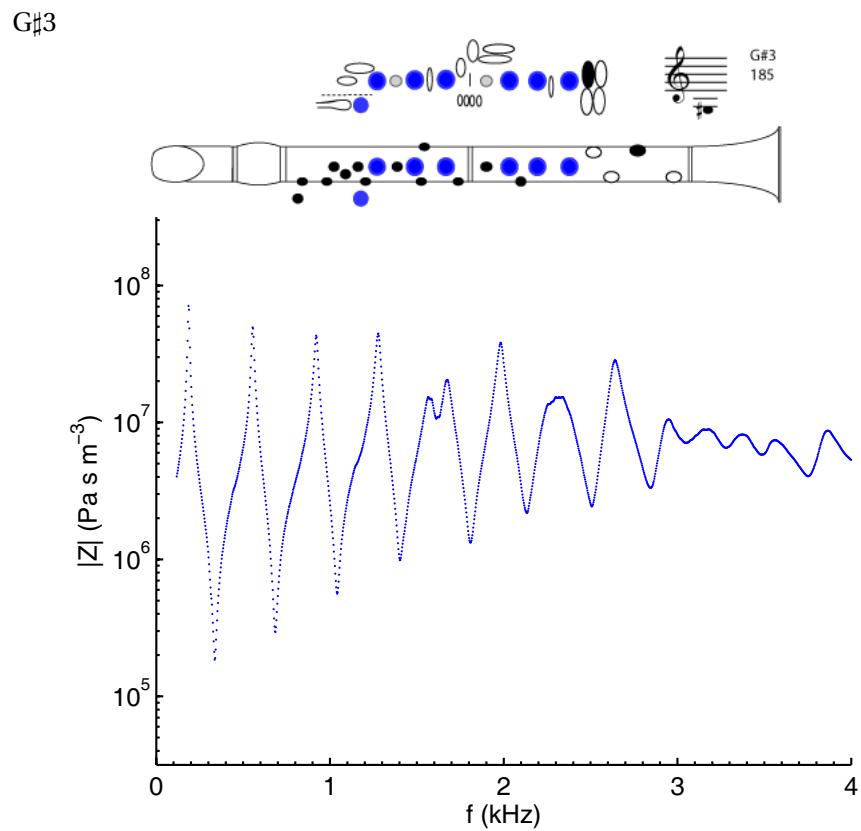


F#3

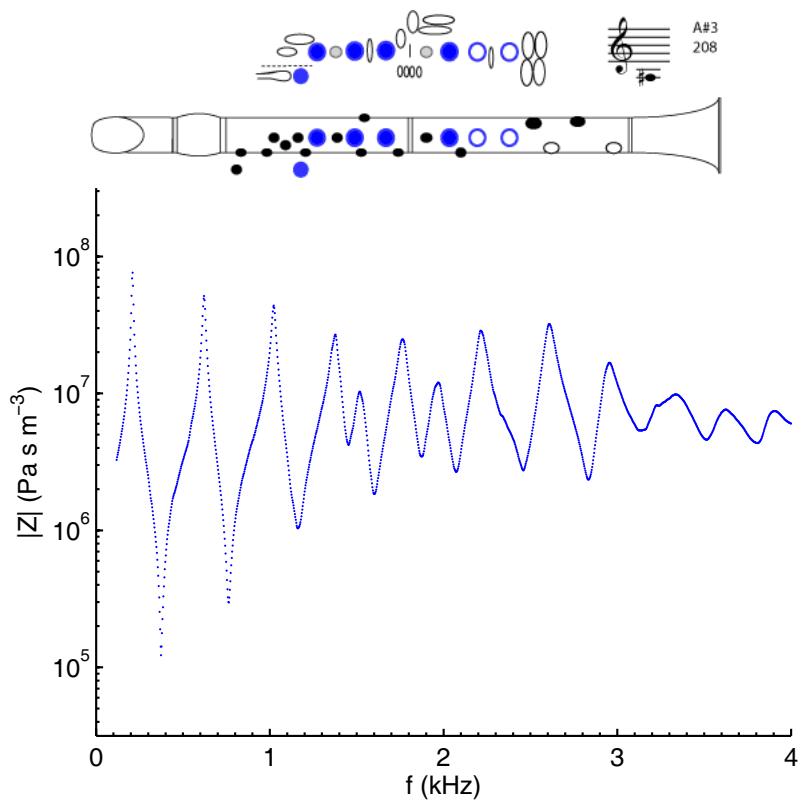


G3

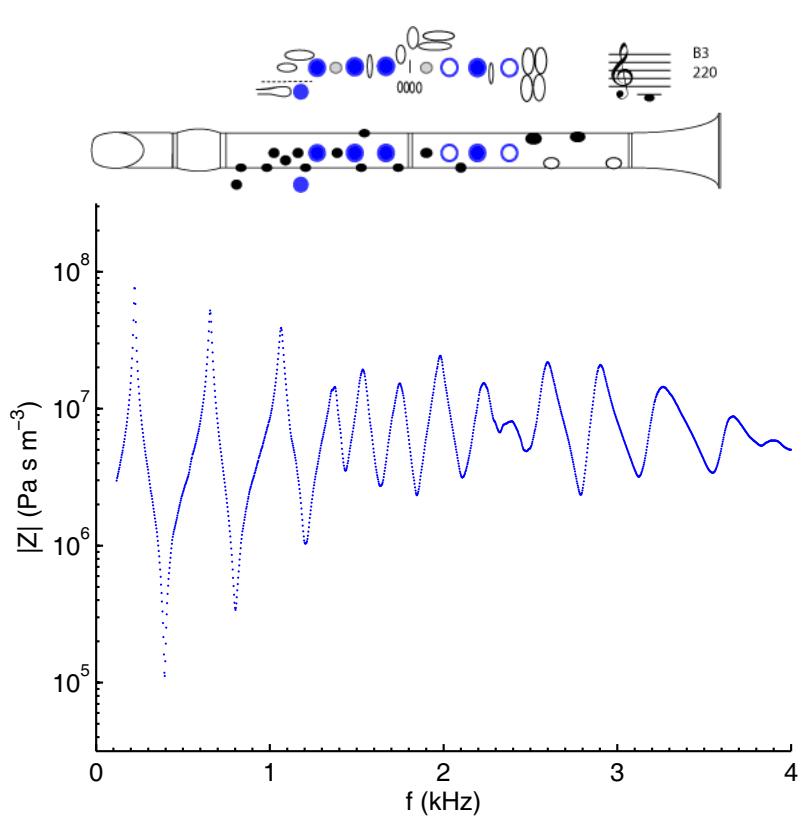




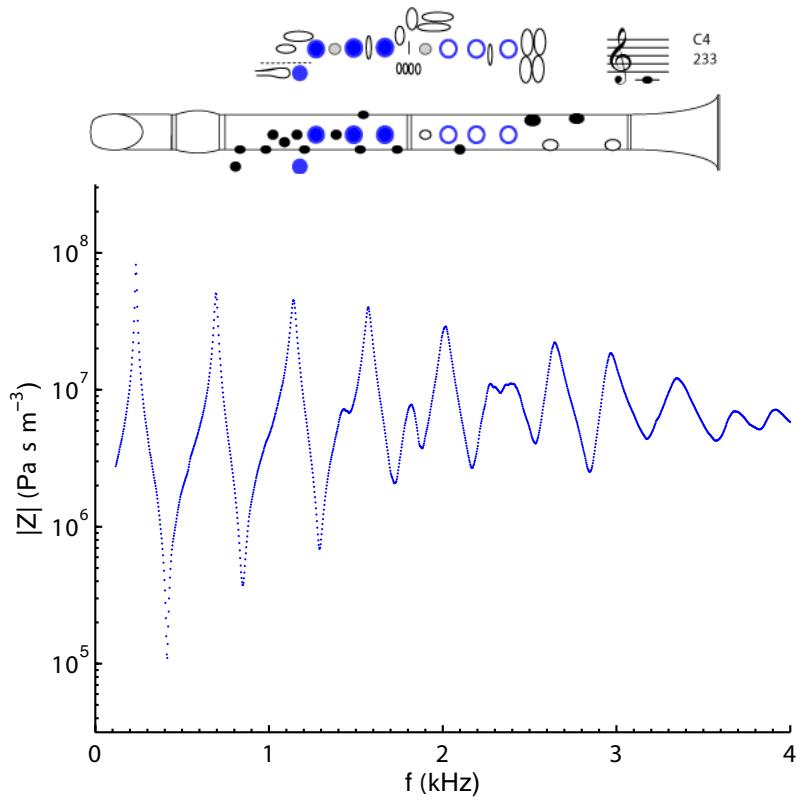
A#3



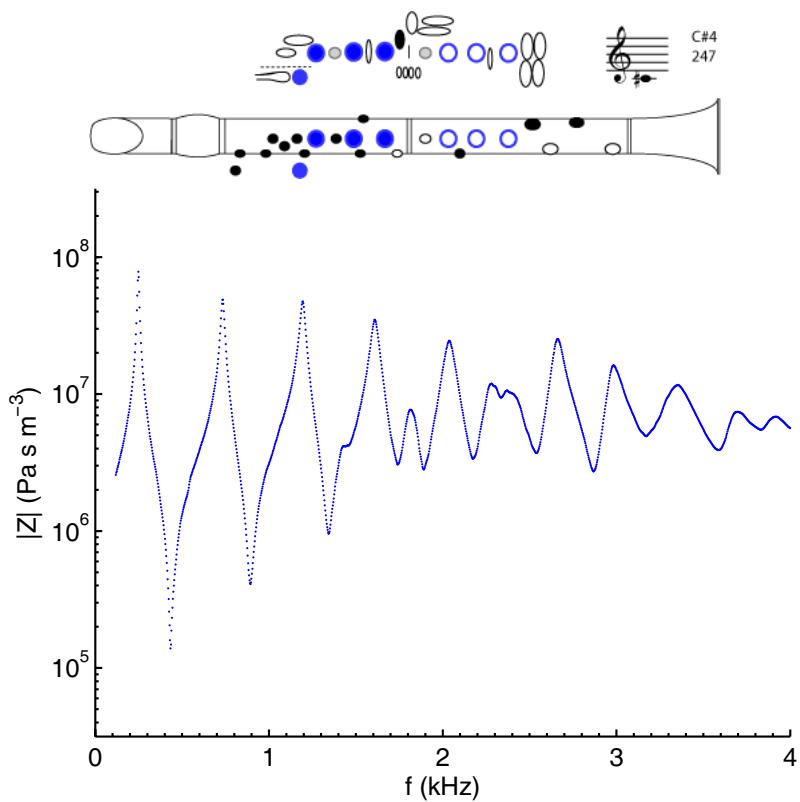
B3



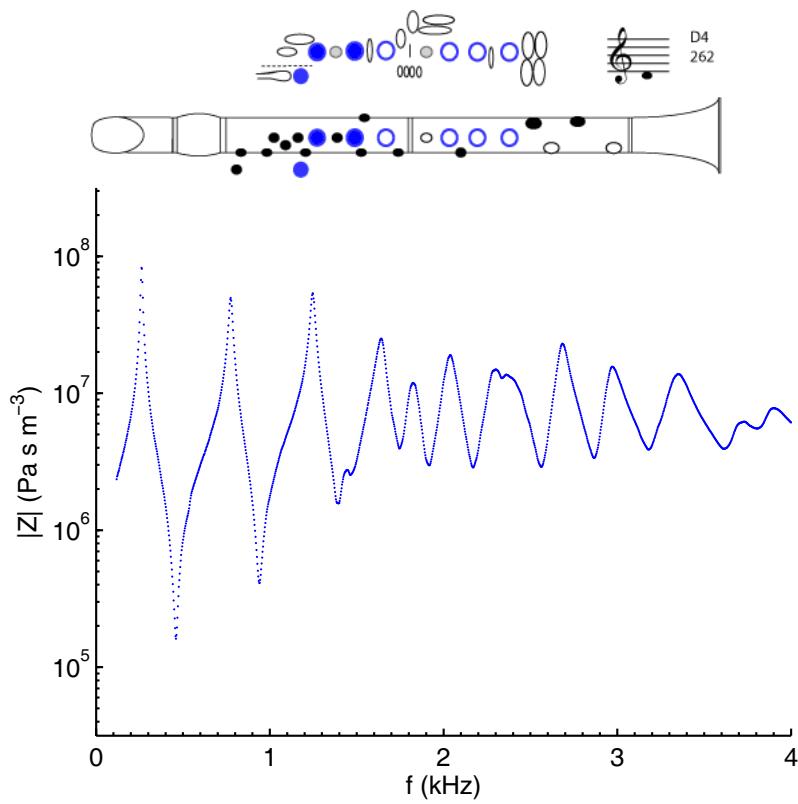
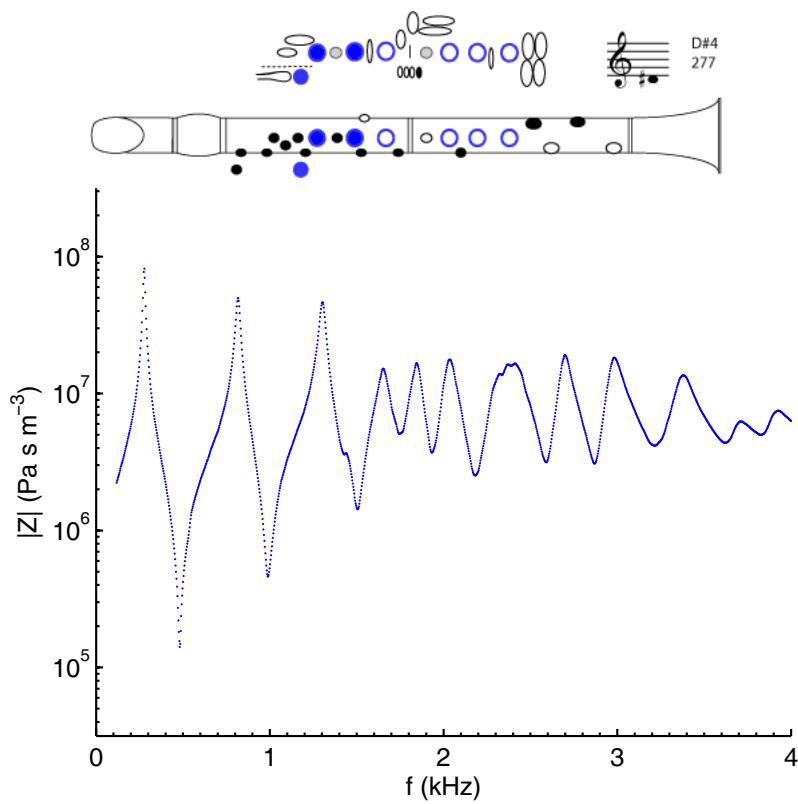
C4



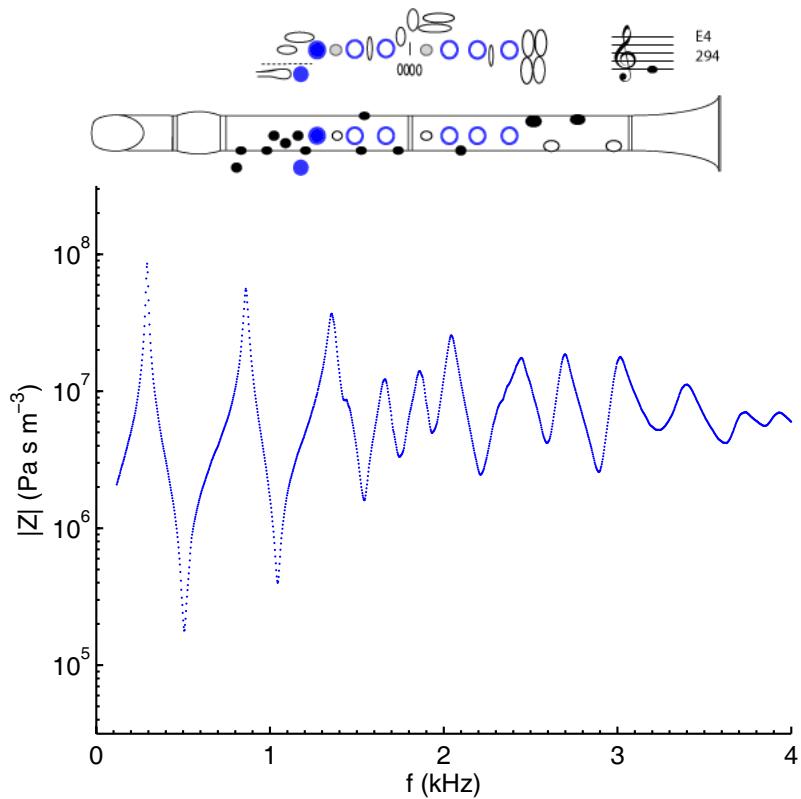
C♯4



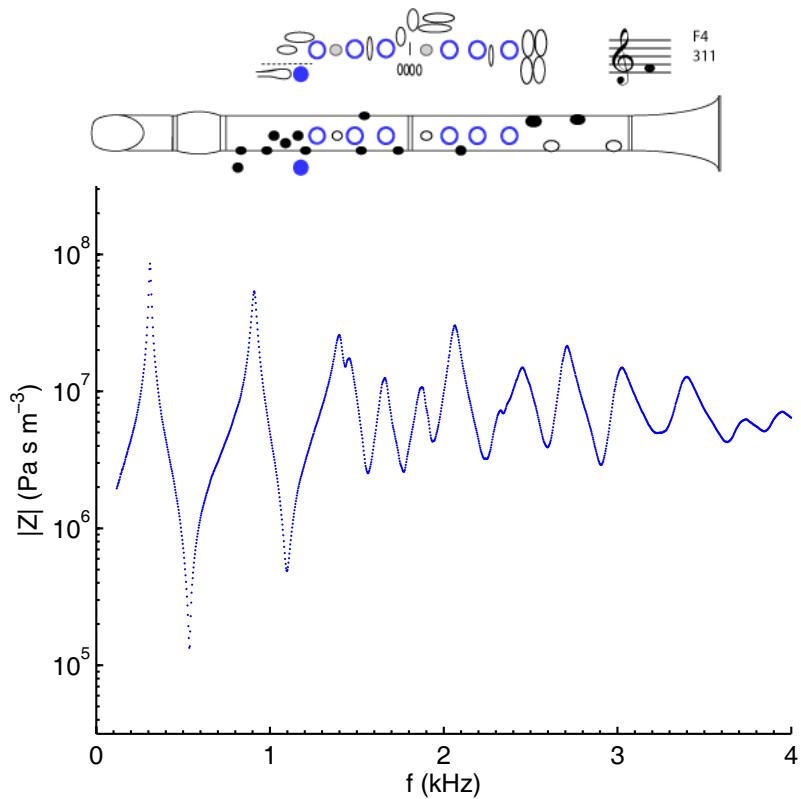
D4

D \sharp 4

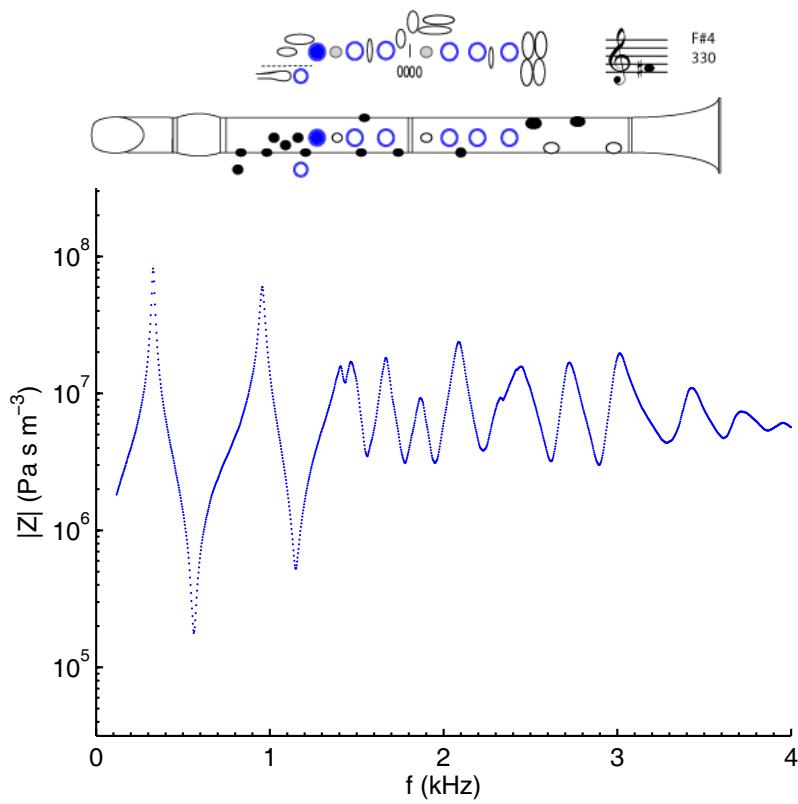
E4



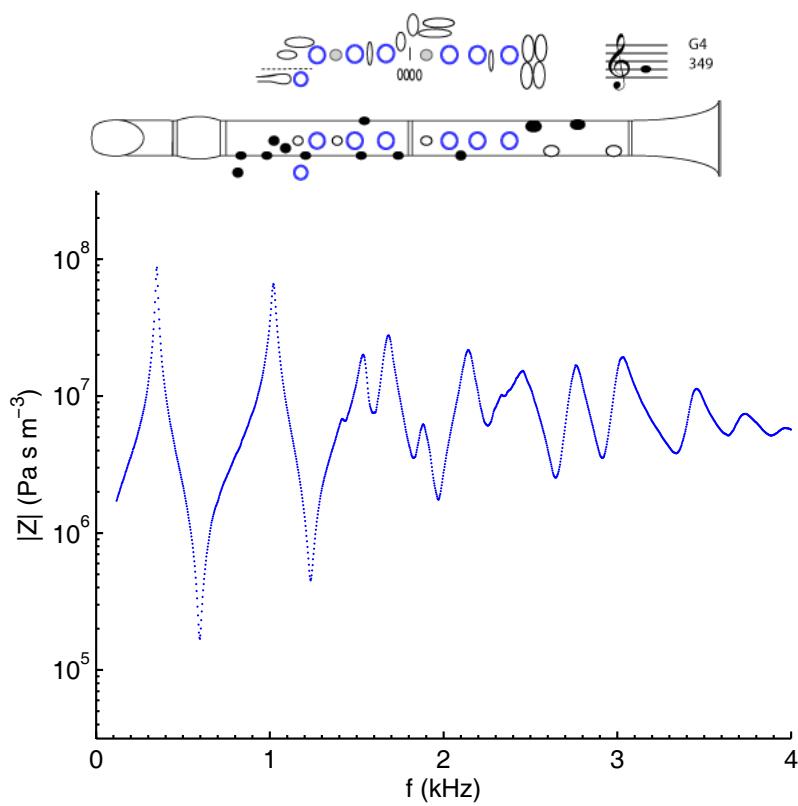
F4

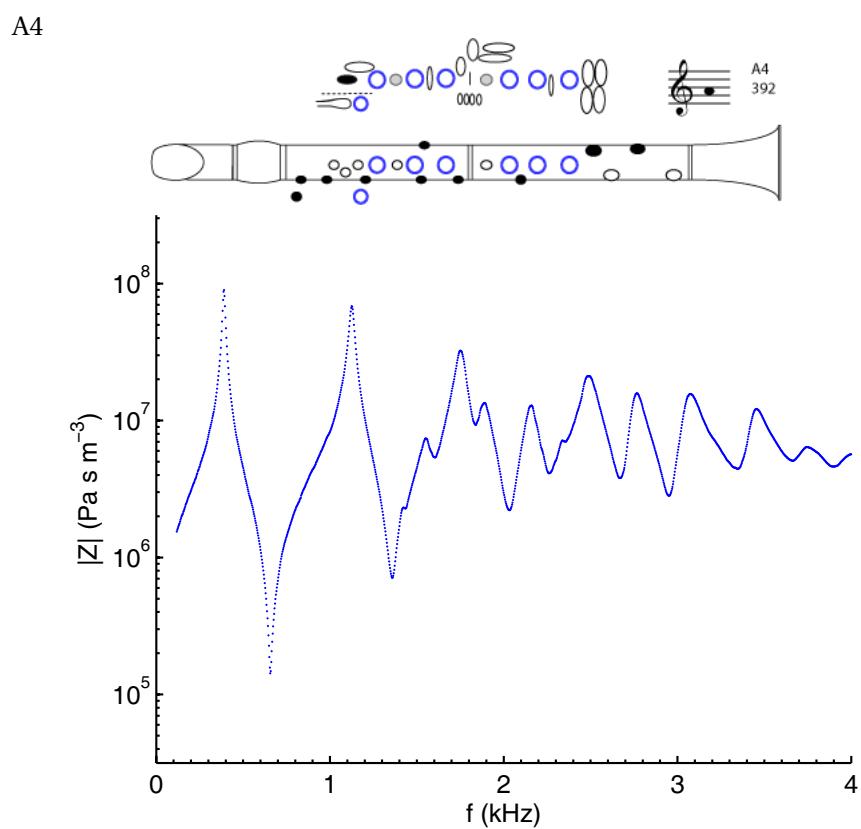
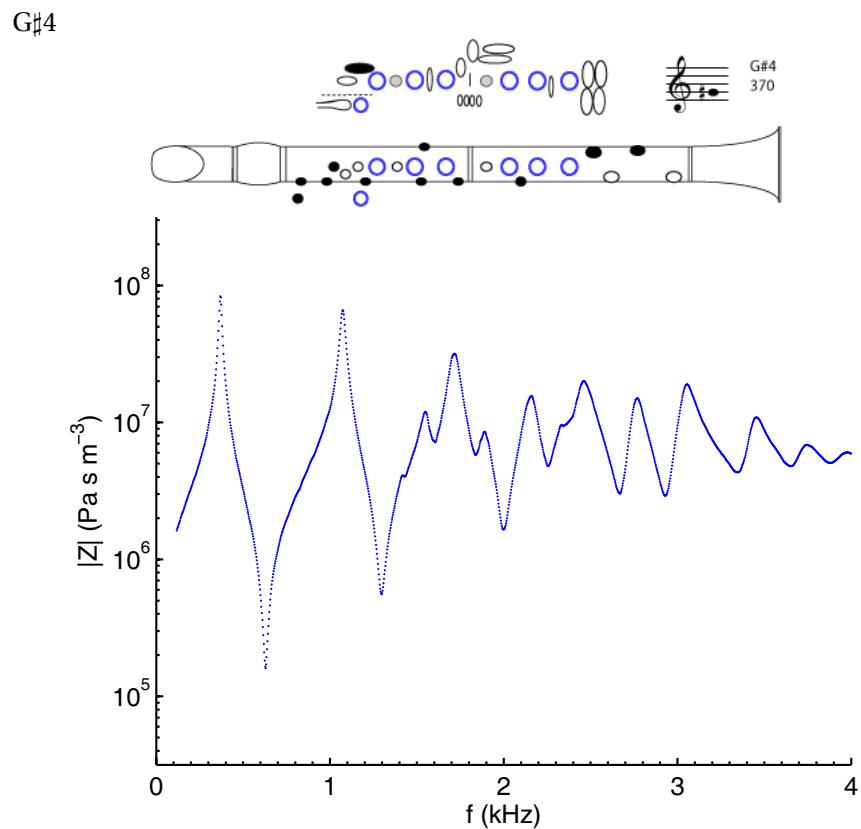


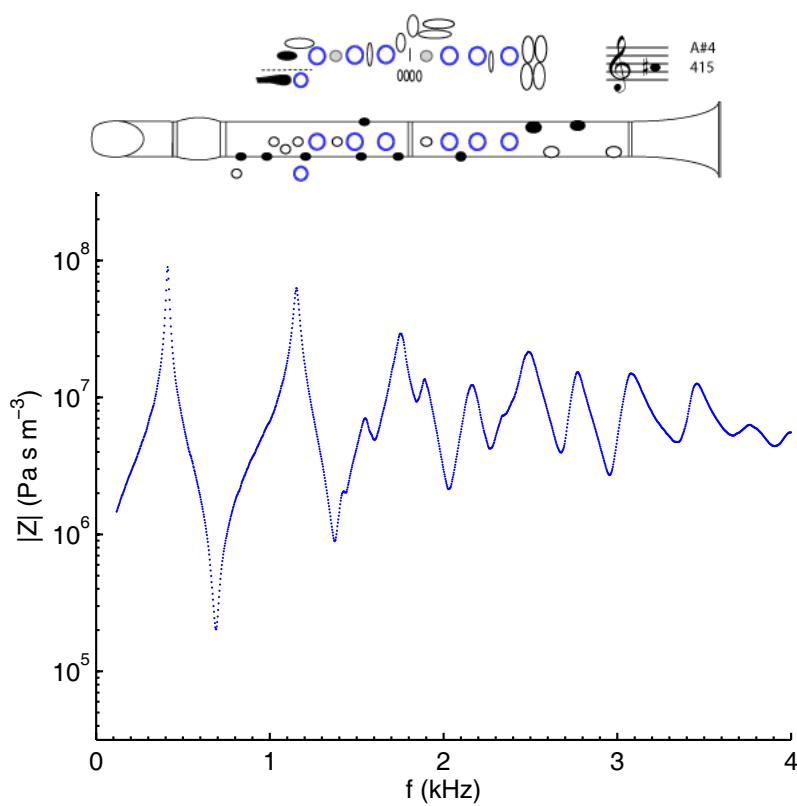
F#4



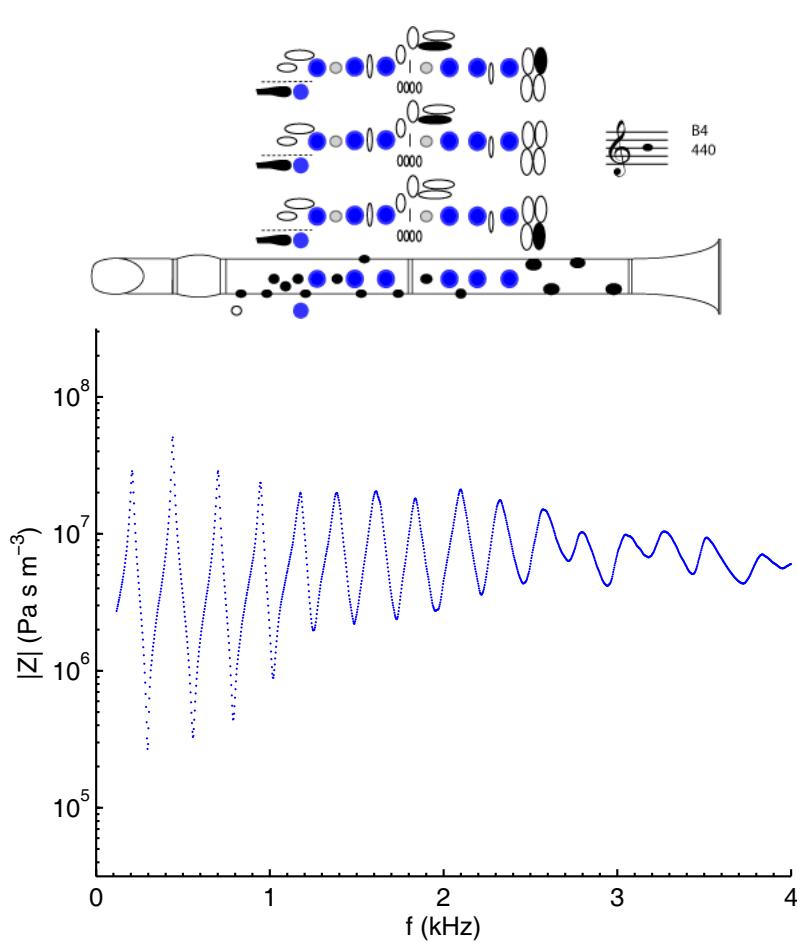
G4

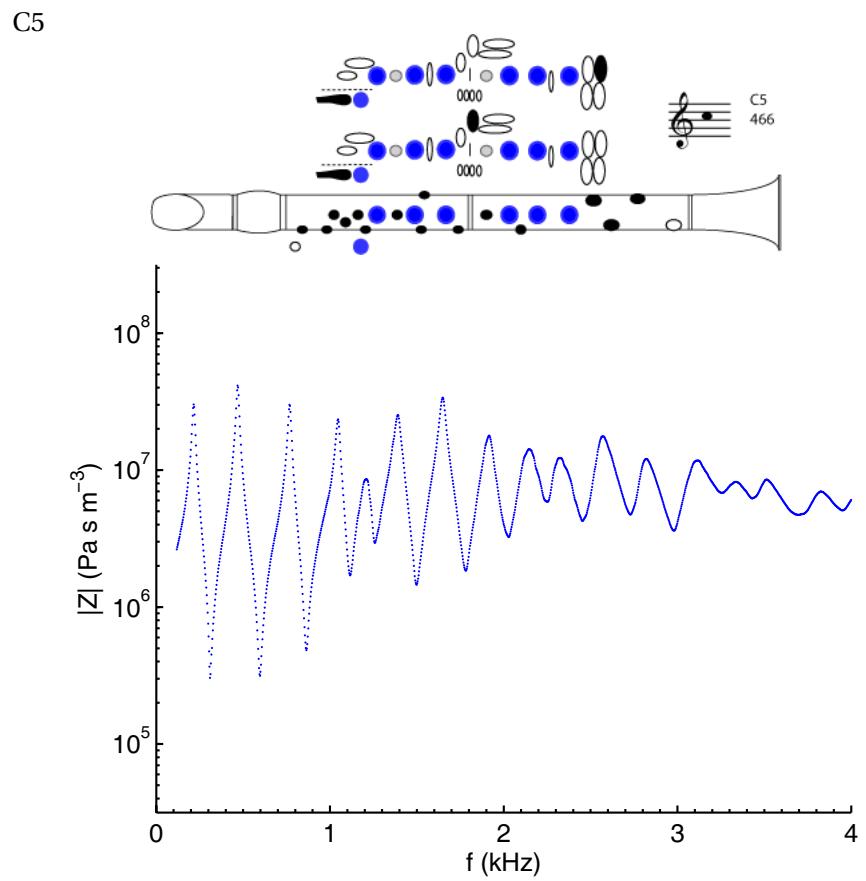




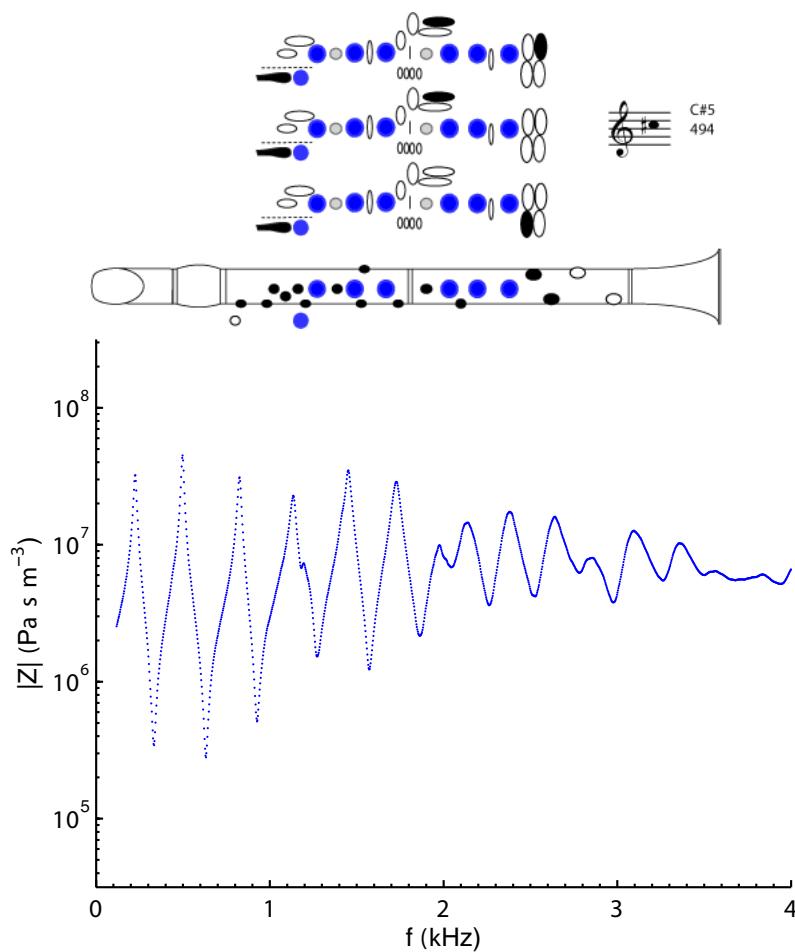
A[#]4

B4

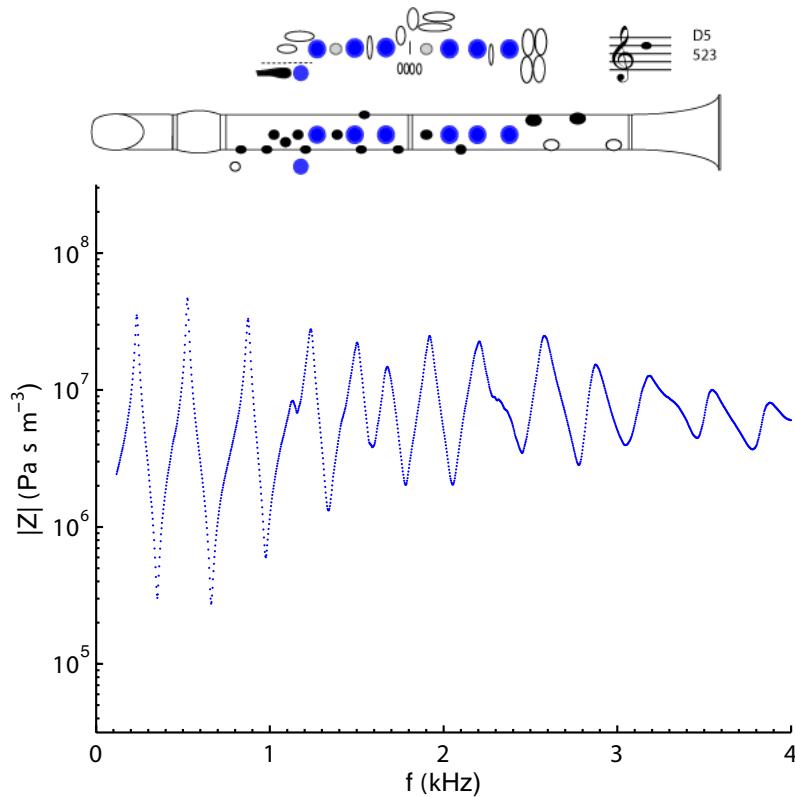


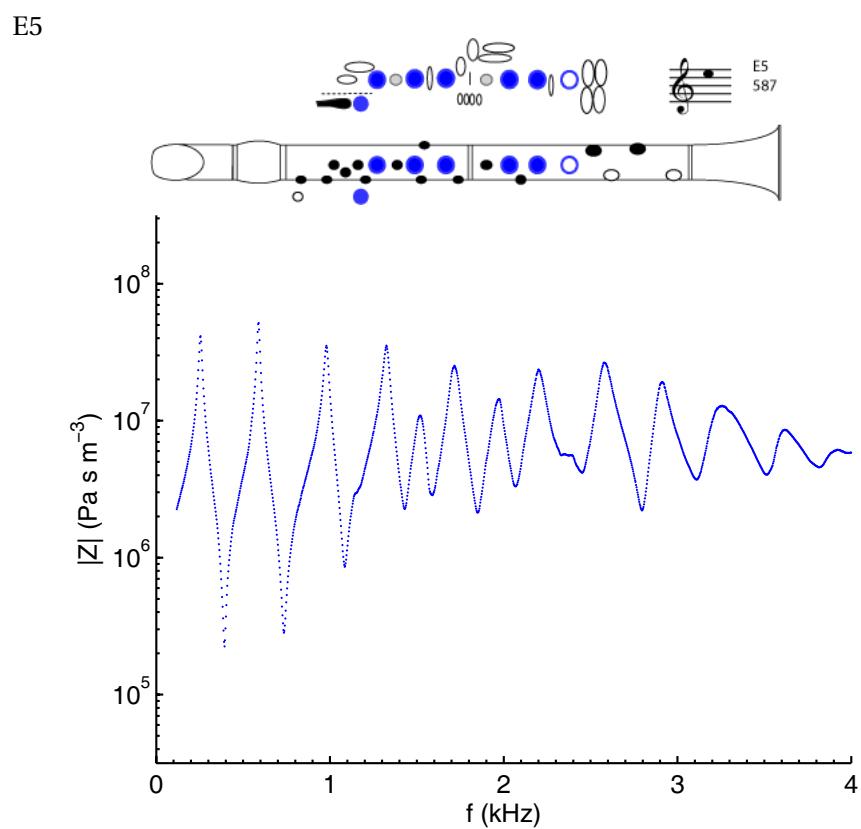
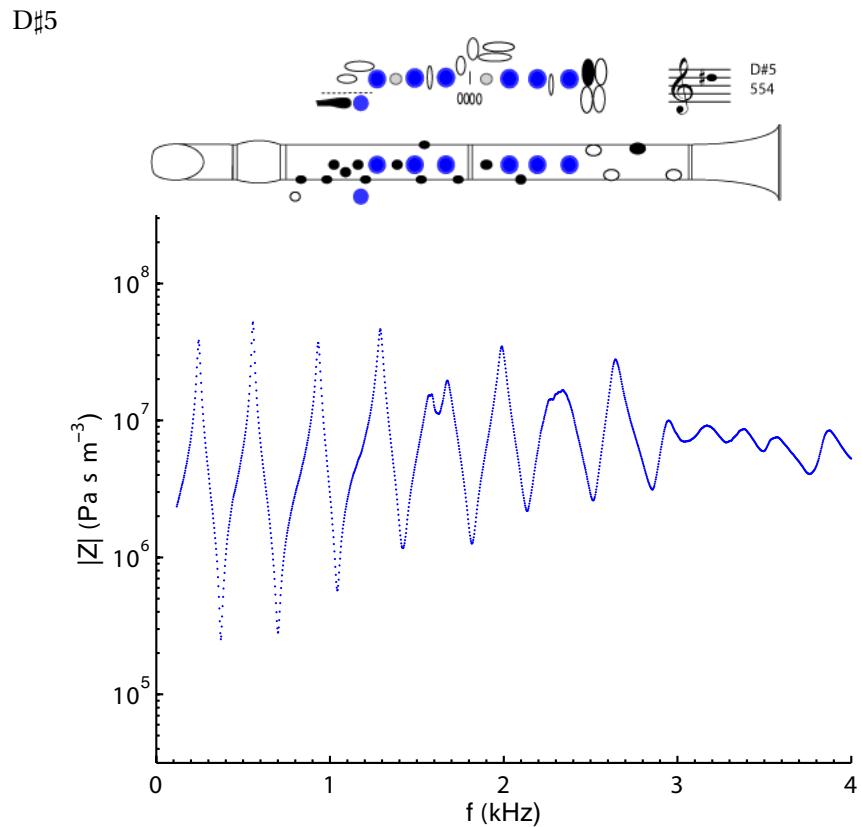


C♯5

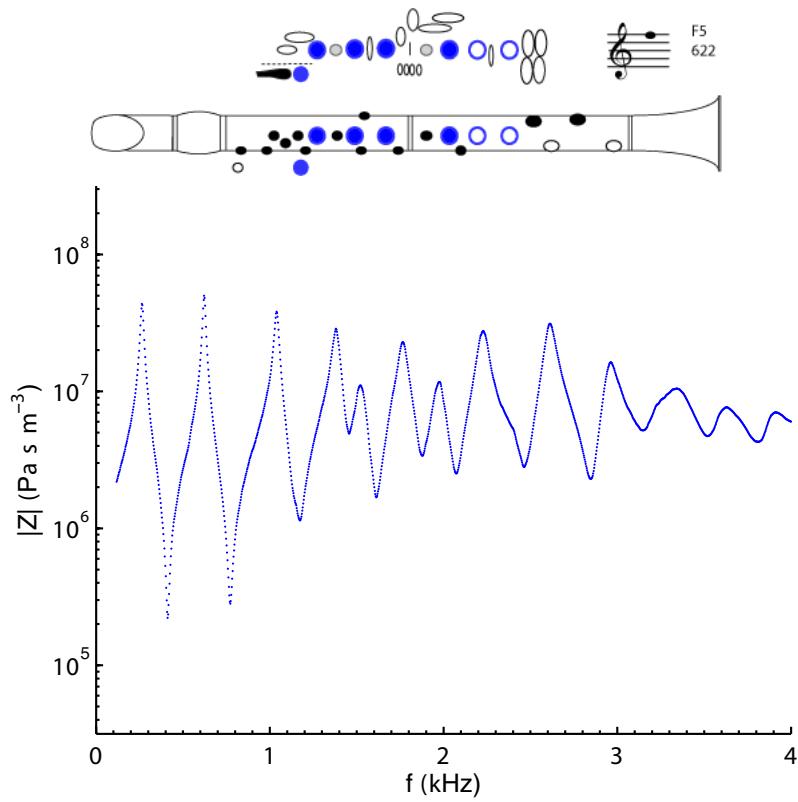


D5

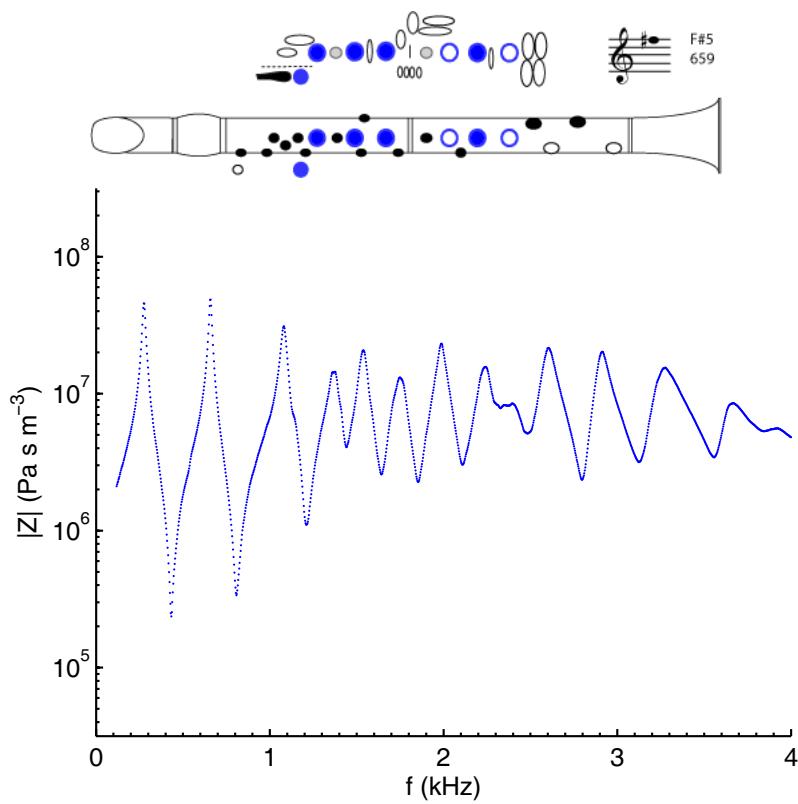


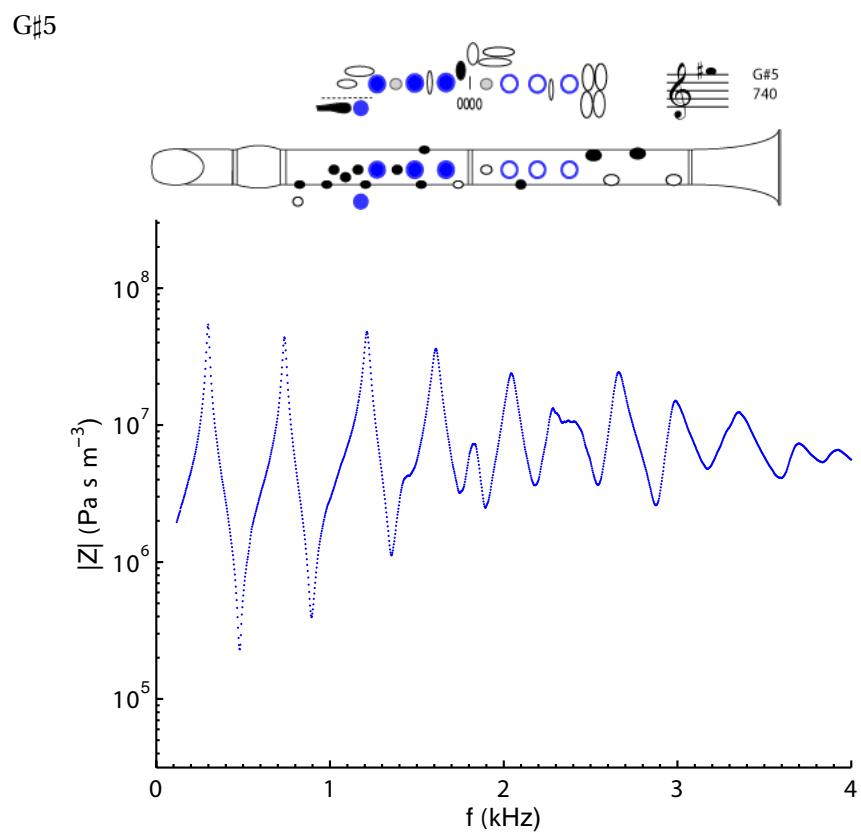
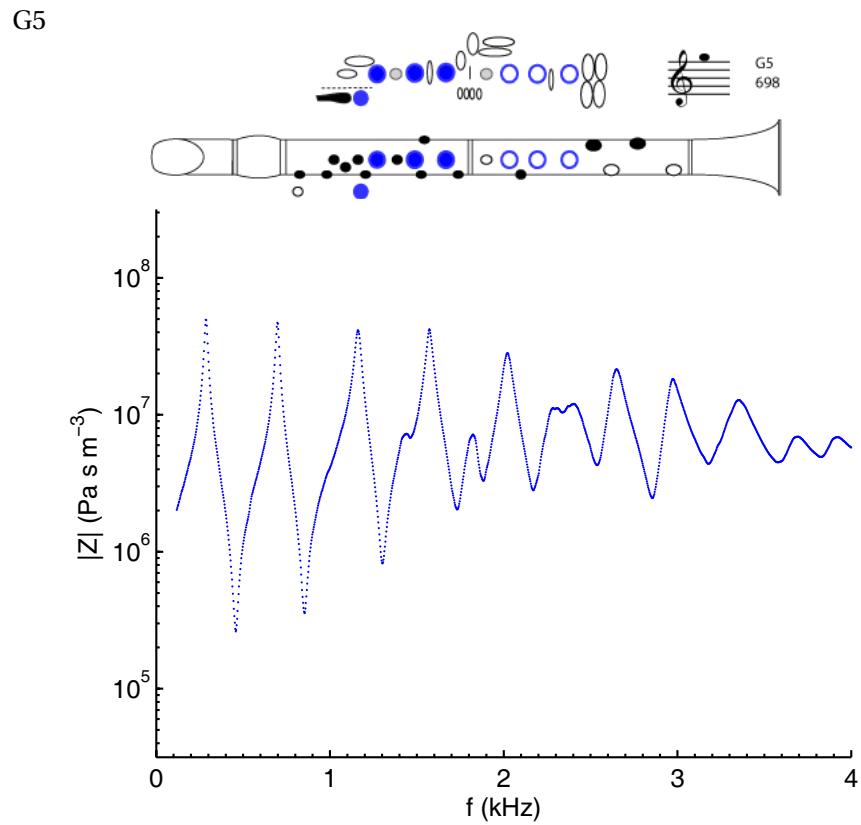


F5

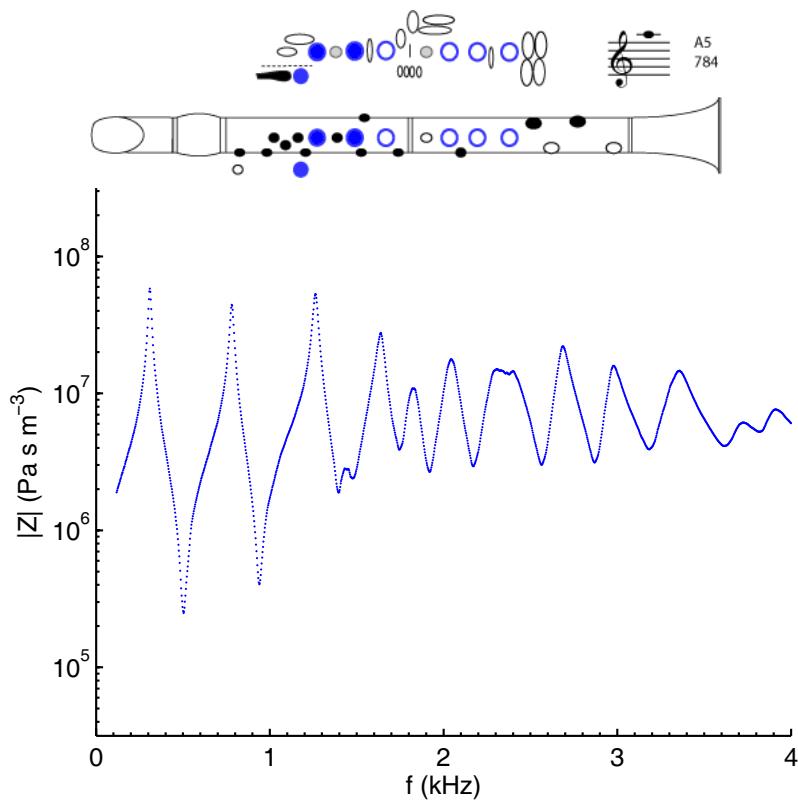


F#5

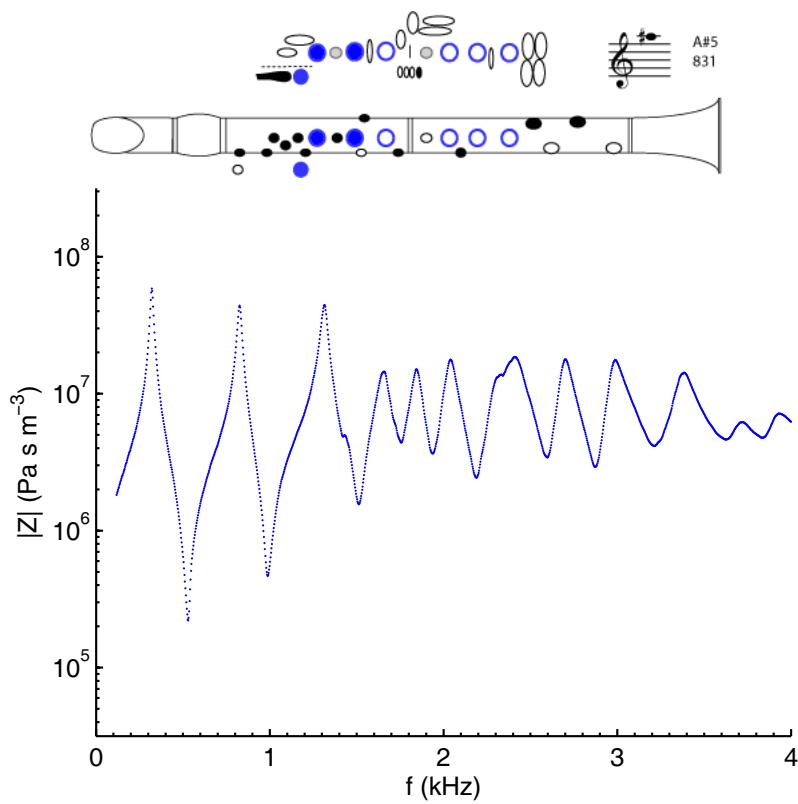




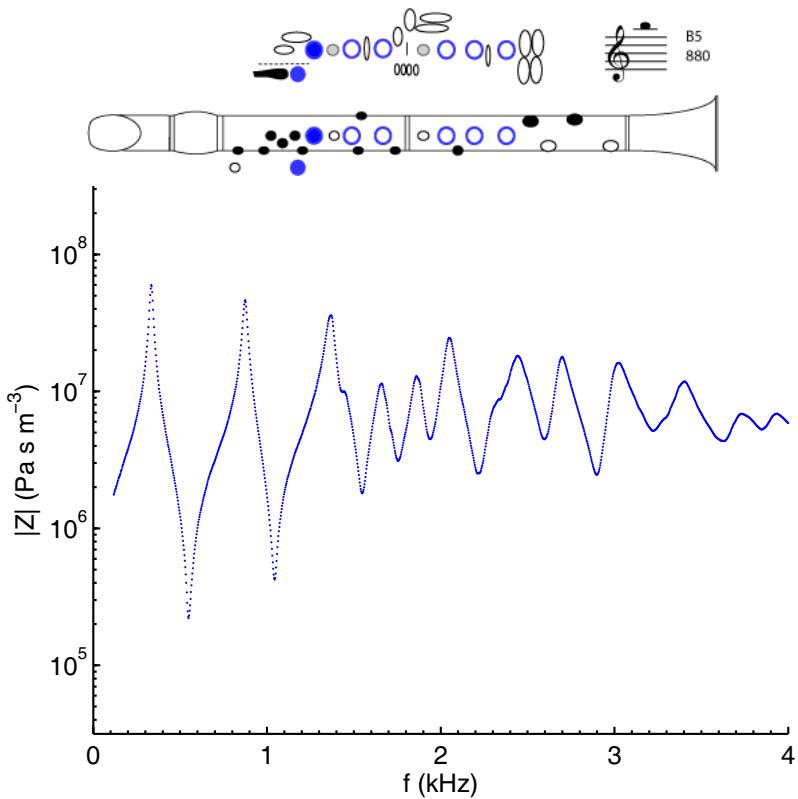
A5



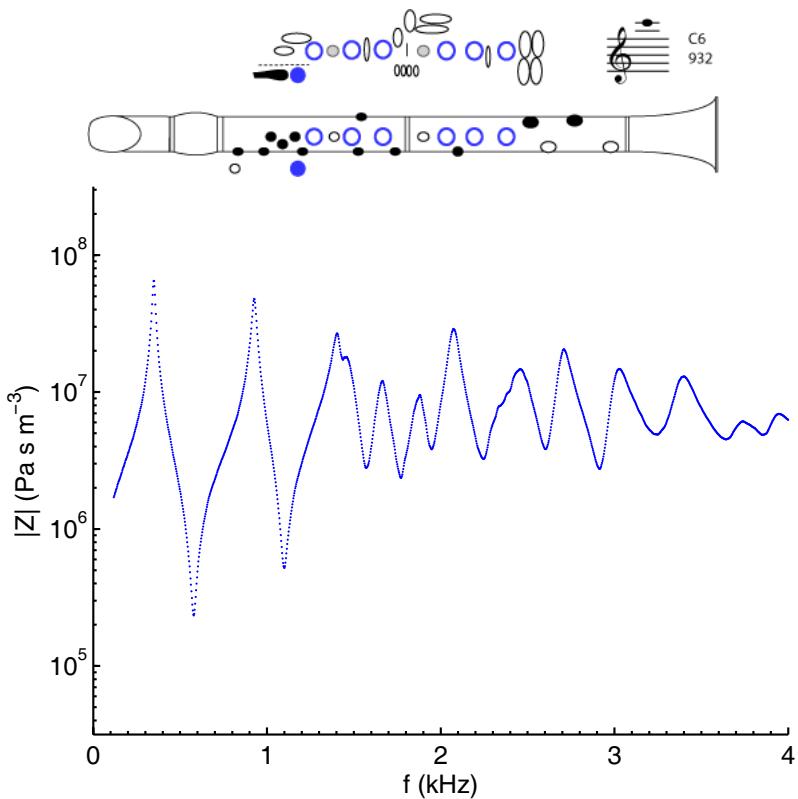
A♯5

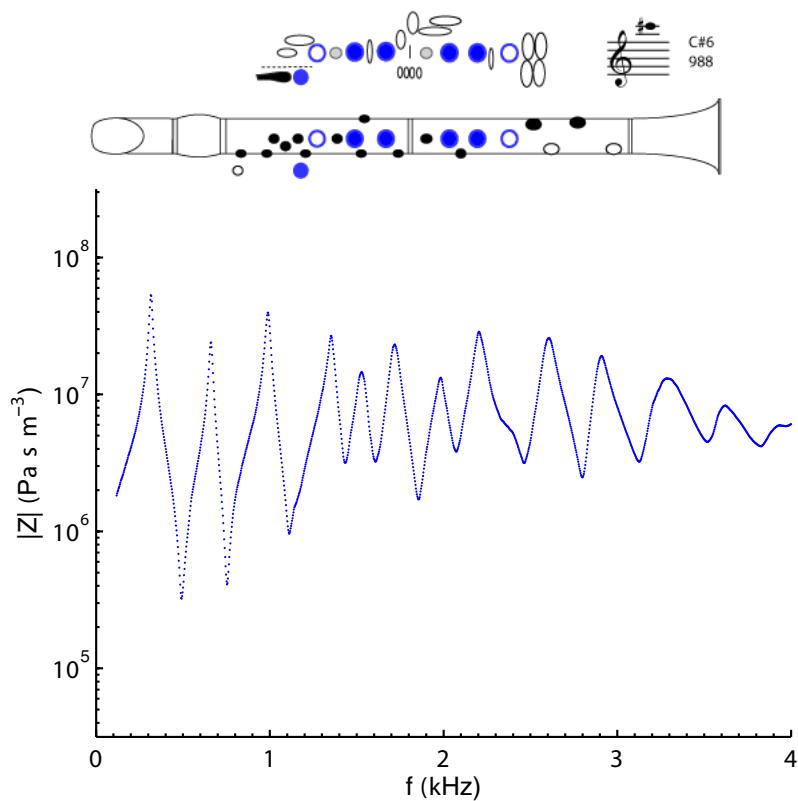


B5

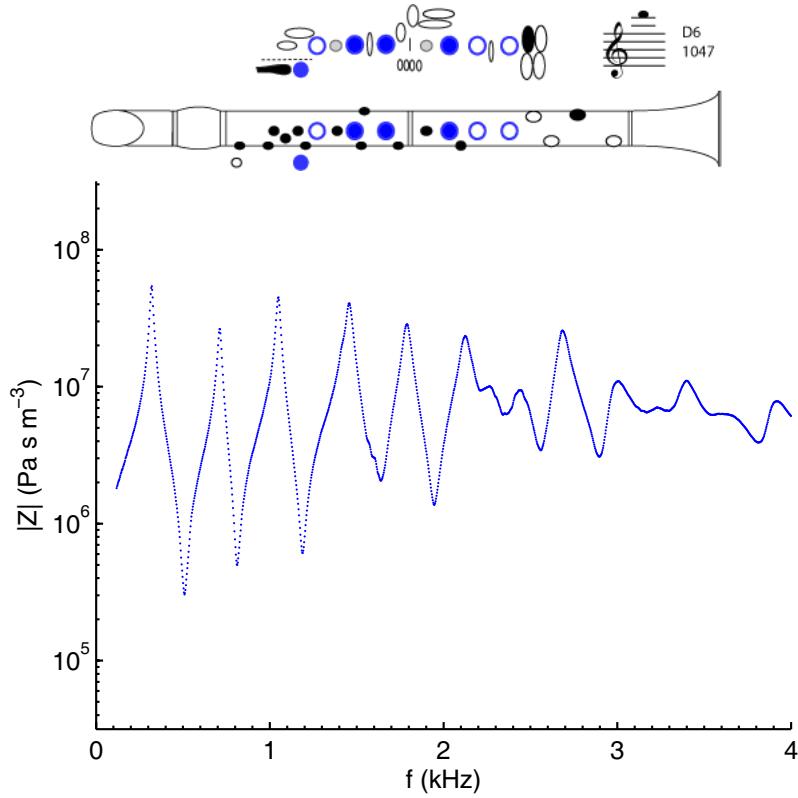


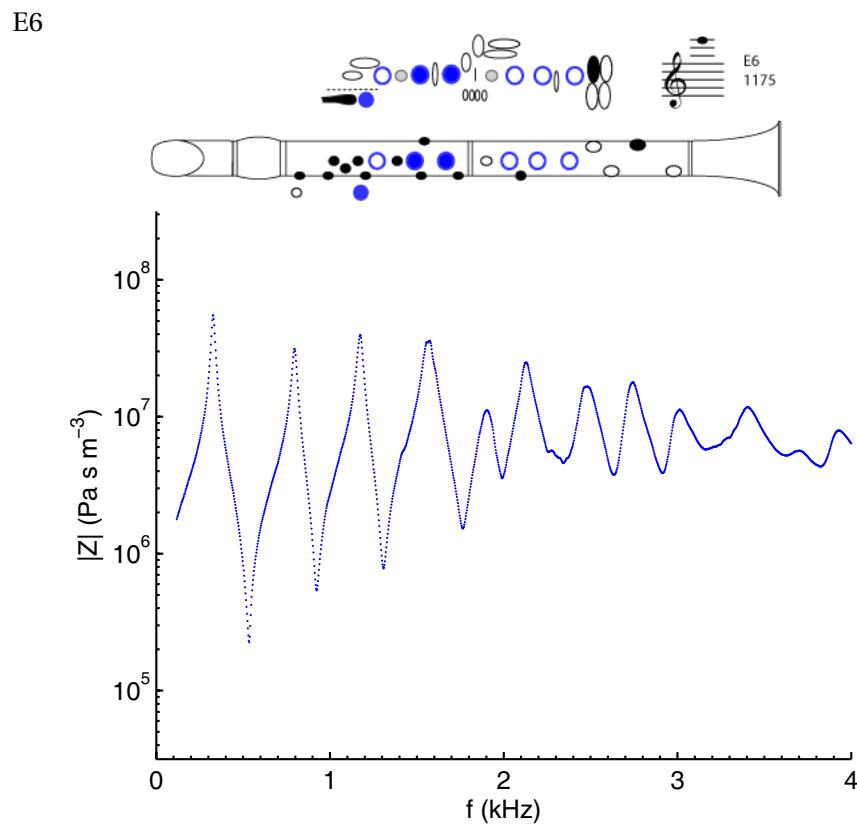
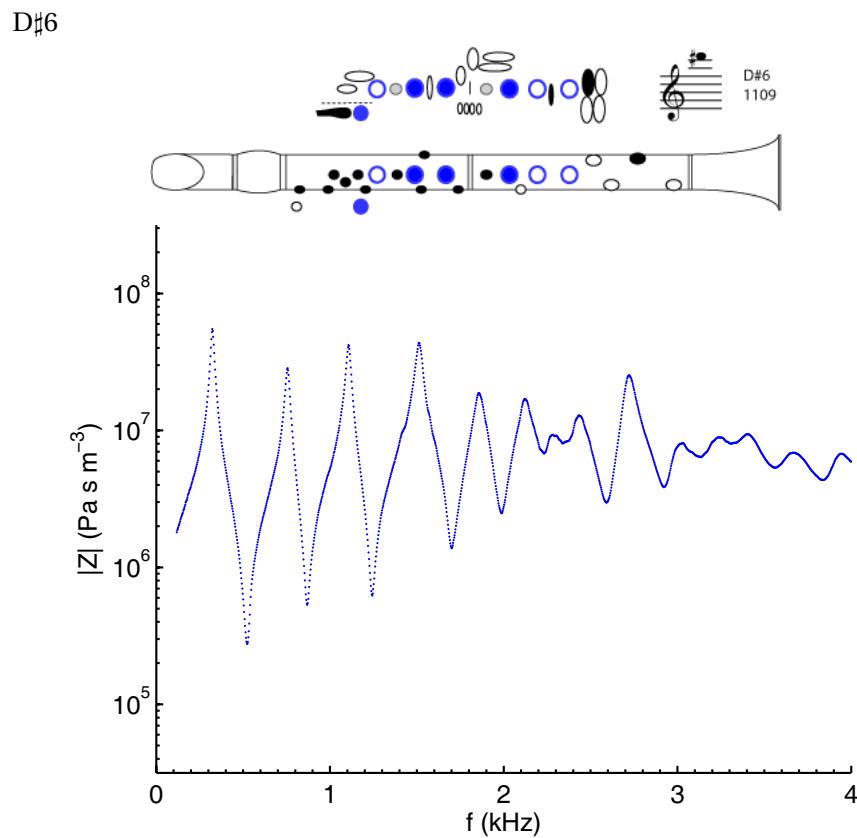
C6



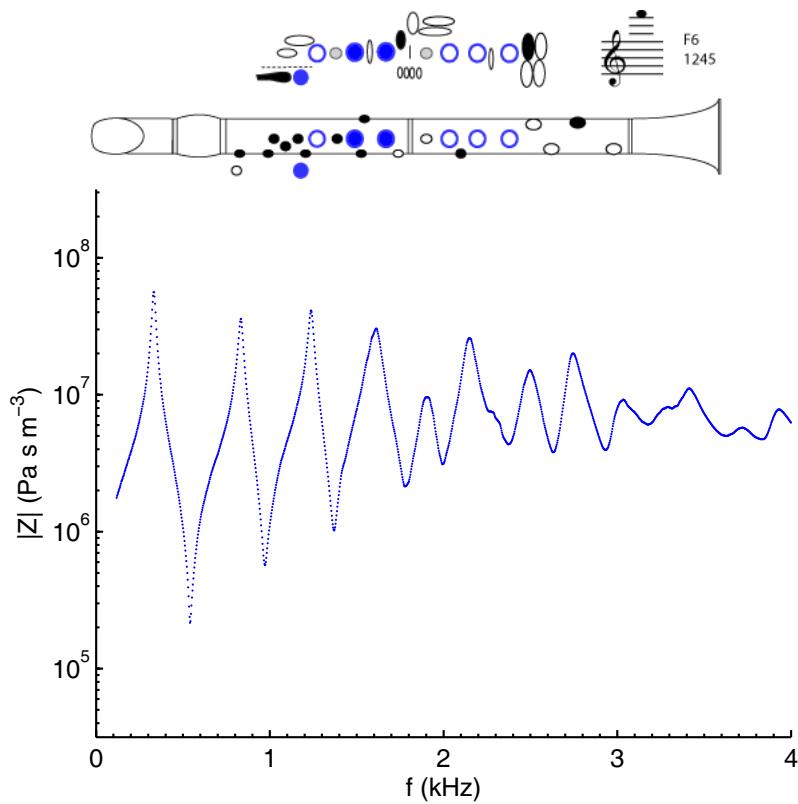
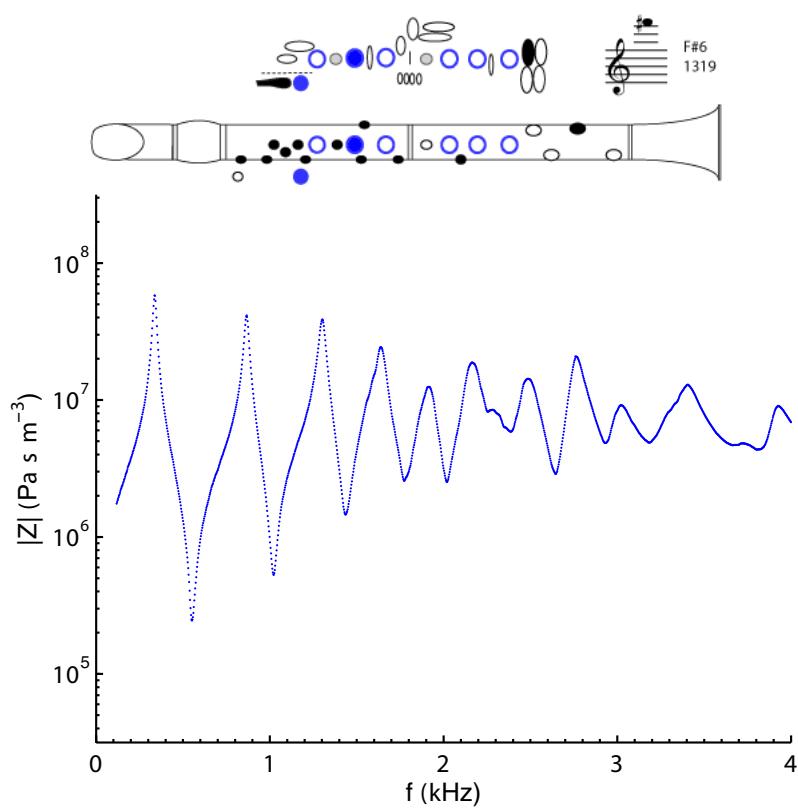
C \sharp 6

D6

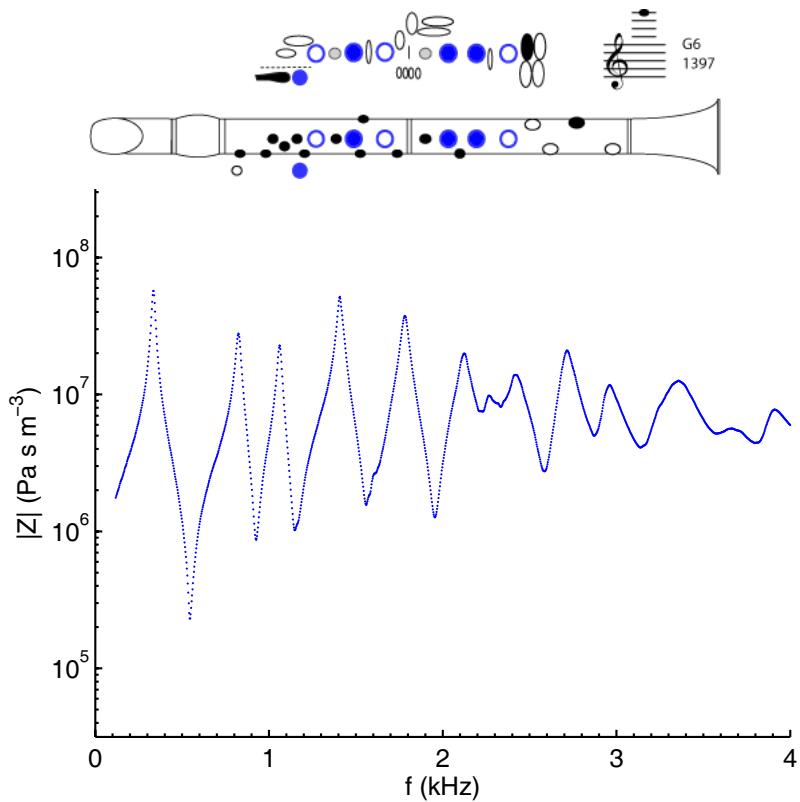




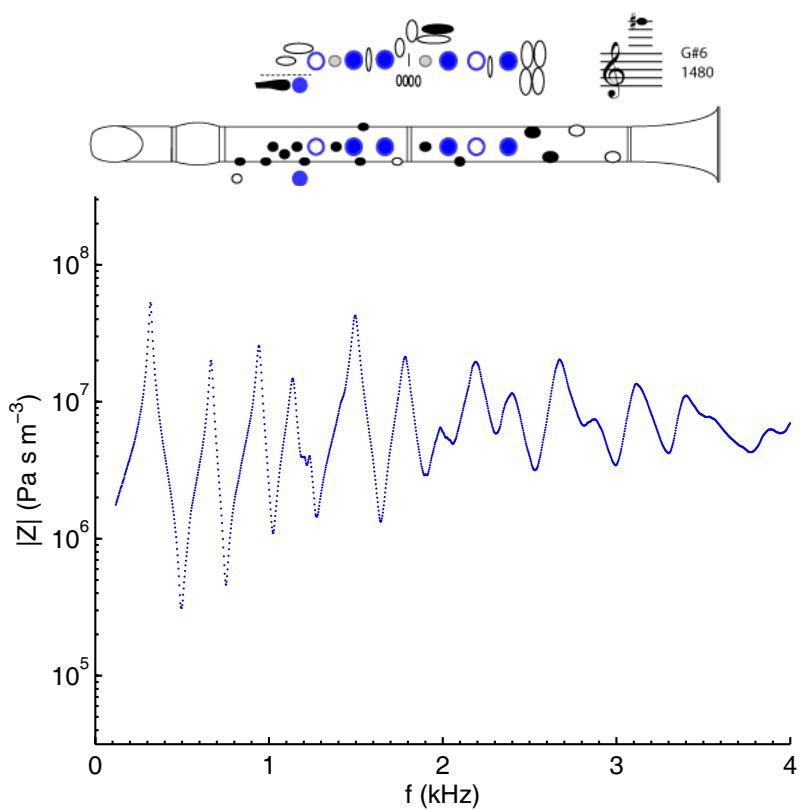
F6

F \sharp 6

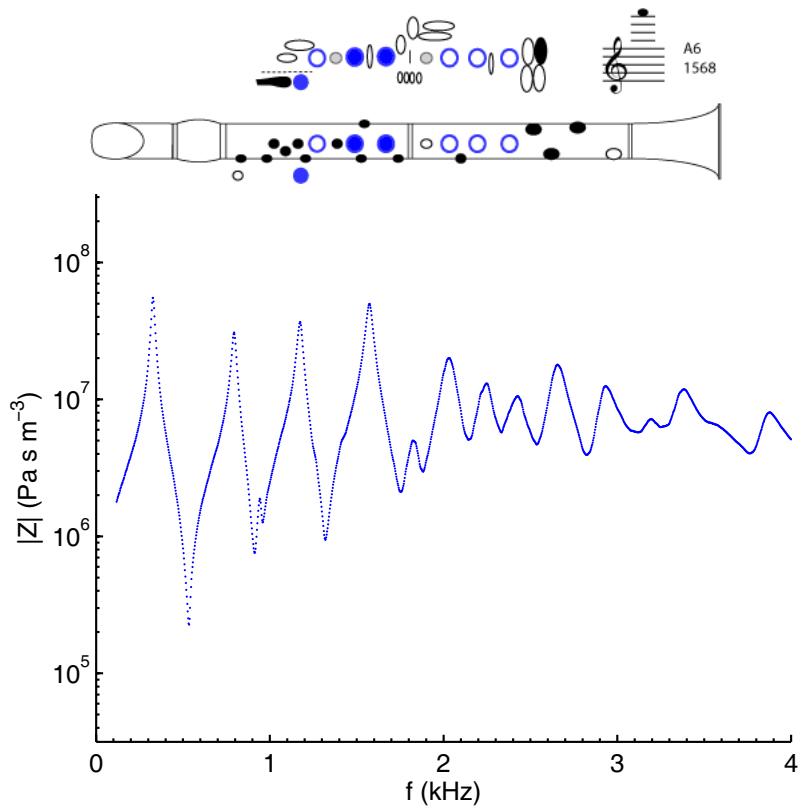
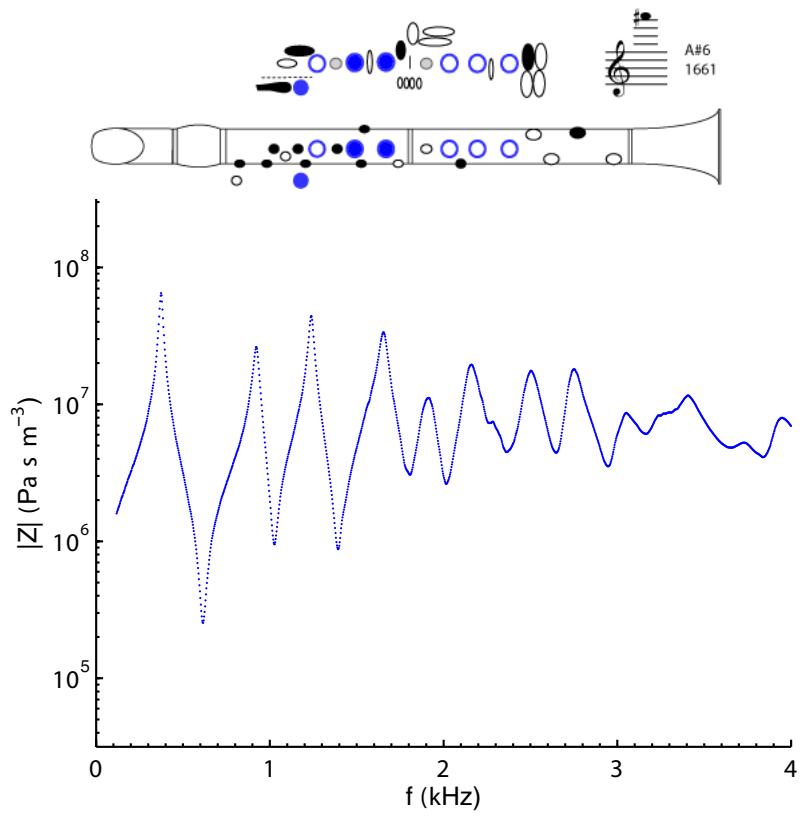
G6



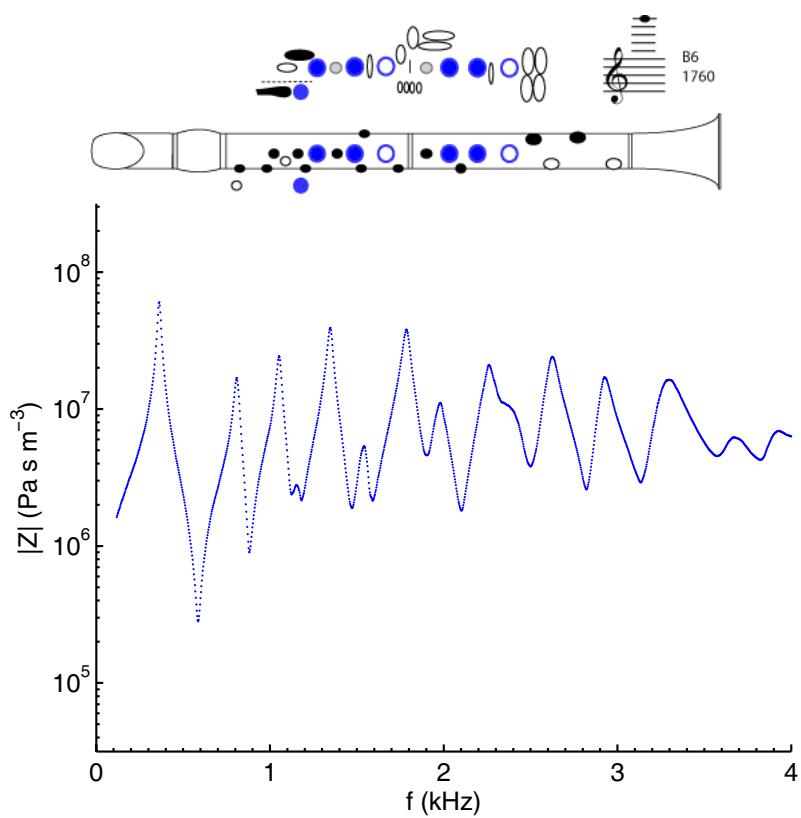
G♯6



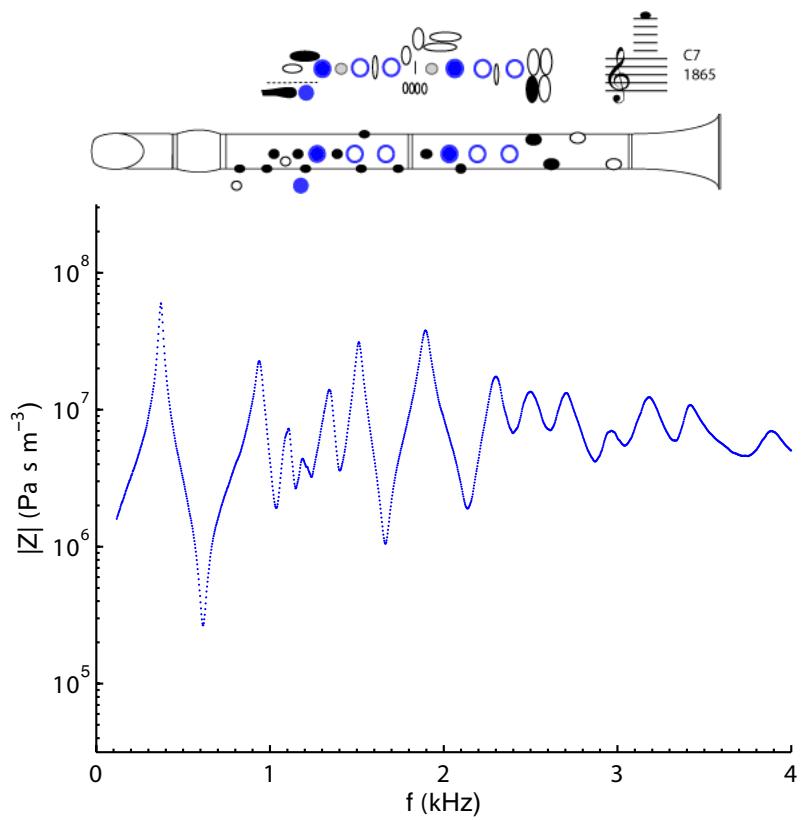
A6

A \sharp 6

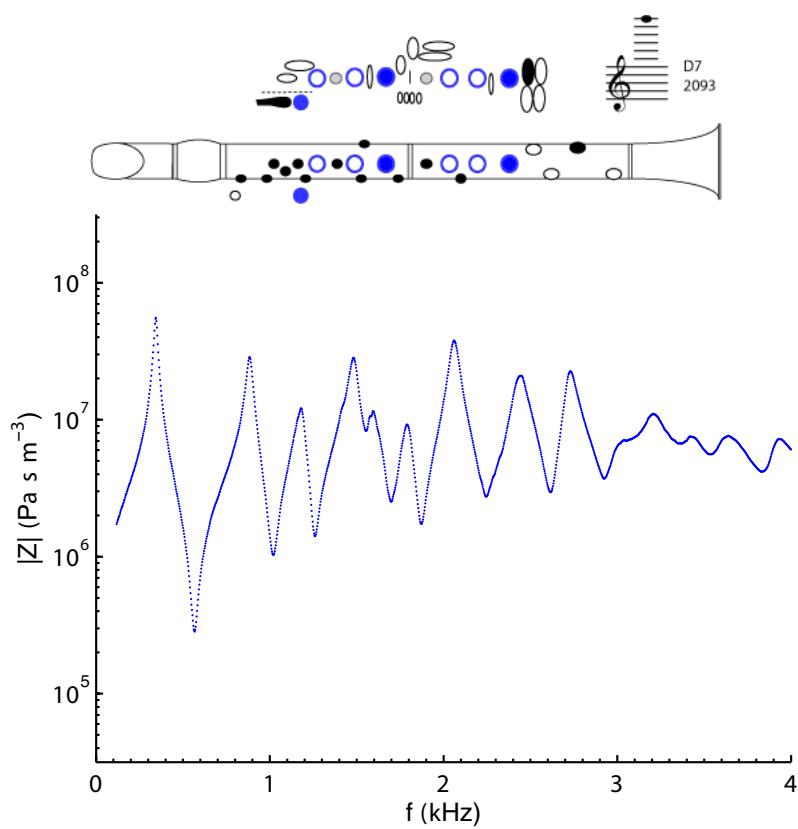
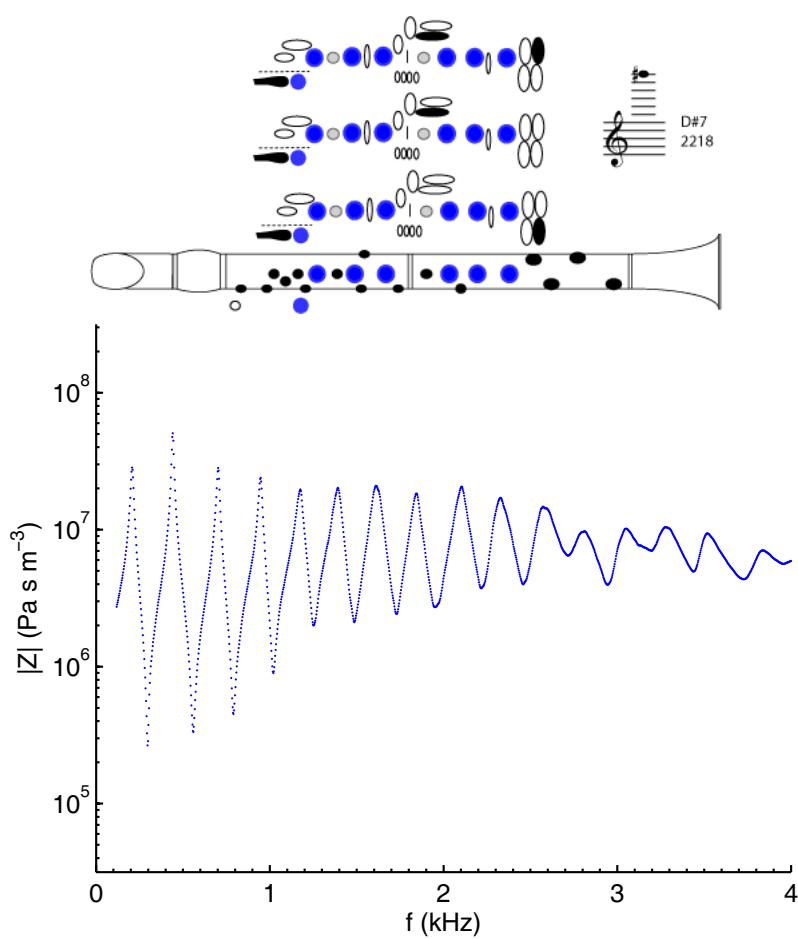
B6

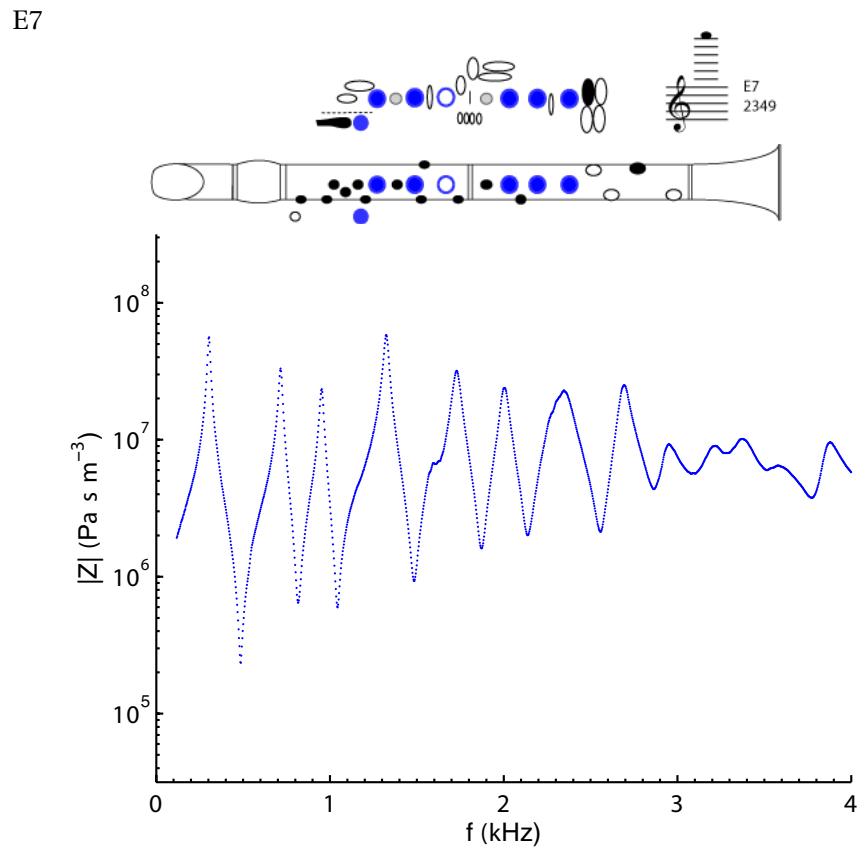


C7



D7

D \sharp 7



Appendix B

Program listings

B.1	Acoustics.h	196
B.2	Acoustics.c	201
B.3	AnalyseNotes.c	207
B.4	Analysis.h	209
B.5	Analysis.c	213
B.6	BbClarinet.xml	223
B.7	ClassicalFlute.xml	231
B.8	Complex.h	235
B.9	Complex.c	239
B.10	Impedance.c	242
B.11	Map.h	246
B.12	Map.c	248
B.13	Minima.h	250
B.14	Minima.c	255
B.15	ModernFlute.xml	268
B.16	Note.h	275
B.17	Note.c	276
B.18	ParseImpedance.h	279
B.19	ParseImpedance.c	280
B.20	ParseXML.h	282
B.21	ParseXML.c	287
B.22	PlayedImpedance.c	293
B.23	Point.h	297
B.24	Point.c	298
B.25	TransferMatrix.h	299
B.26	TransferMatrix.c	301
B.27	Vector.h	303
B.28	Vector.c	305
B.29	Waves.c	308
B.30	Woodwind.h	311
B.31	Woodwind.c	322
B.32	Woodwind.dtd	336

Listing B.1: Acoustics.h

```
/*
Acoustics.h
By Paul Dickens, 2005

Acoustics library.
*/



#ifndef ACOUSTICS_H_PROTECTOR
#define ACOUSTICS_H_PROTECTOR


#include "Complex.h"
#include "TransferMatrix.h"


double saturationVapourPressureWater(double T);
/*
Calculates the saturation vapour pressure of water.
Parameters:
T: the temperature in Kelvin.
Returns:
The vapour pressure in Pa.
*/


double speedSound(double t, double p, double h, double x_c);
/*
Calculates the speed of sound.
Based on Owen Cramer (1993) J. Acoust. Soc. Am. 93(5) p2510-2616;
formula at p. 2514
Parameters:
t: the temperature in Celsius (between 0 and 30 degrees C).
h: the relative humidity expressed as a fraction (between 0 and 1).
x_c: the mole fraction of carbon dioxide (between 0 and 0.01).
(note that the parameter ranges are not enforced)
(note that Cramer gives x_c = 0.000314 for air)
Returns:
The speed of sound in m/s.
*/


double speedSoundCramer(double t, double p, double x_w, double x_c);
/*
Calculates the speed of sound.
Based on Owen Cramer (1993) J. Acoust. Soc. Am. 93(5) p2510-2616;
formula at p2514.
Parameters:
t: the temperature in Celsius (between 0 and 30 degrees C).
p: the pressure in Pa (between 75 and 102 kPa).
x_w: the mole fraction of water vapour (between 0 and 0.06).
x_c: the mole fraction of carbon dioxide (between 0 and 0.01).
(note that the parameter ranges are not enforced)
Returns:
The speed of sound in m/s.
*/


double densityAirIdeal(double t, double p, double h, double x_c);
```

```

/*
  Calculates the density of air using the ideal gas law.
  For a more accurate treatment, see Giacomo (1982), Metrologia 18,
  33--40.
  Parameters:
    t: the temperature in Celsius.
    p: the pressure in Pa.
    h: the relative humidity (between 0 and 1).
    x_c: the mole fraction of carbon dioxide.
  Returns:
    The density of air in kg/m3.
 */

double densityAirGiacomo(double t, double p, double h, double x_c);
/*
  Calculates the density of air.
  Based on Giacomo (1982), Metrologia 18, 33--40.
  Parameters:
    t: the temperature in Celsius.
    p: the pressure in Pa.
    h: the relative humidity (between 0 and 1).
    x_c: the mole fraction of carbon dioxide.
  Returns:
    The density of air in kg/m3.
 */

double compressibilityAir(double t, double p, double x_w);
/*
  Calculates the compressibility of air.
  Parameters:
    t: the temperature in Celsius.
    p: the pressure in Pa.
    h: the relative humidity (between 0 and 1).
    x_w: the mole fraction of water vapour.
  Returns:
    The compressibility of air.
 */

complex waveNum(double f, double c, double a, double alphacorrection);
/*
  Calculates k, the complex wave number for a given segment which
  includes wall losses. Refer to Fletcher and Rossing (1998).
  Parameters:
    f: the frequency.
    c: the speed of sound
    a: the radius of the tube in metres.
    alphacorrection: the multiplicative attenuation coefficient
      factor.
  Returns:
    The complex wave number k, including wall losses.
 */

double phaseVel(double f, double c, double a);
/*
  Calculates the phase velocity of a given pipe, part of the wave
  number.
  Refer to Fletcher and Rossing (1998).
  Parameters:

```

```

f: the frequency.
a: the radius of the tube in metres.
Returns:
The phase velocity.
*/
double attenCoeff(double f, double a, double alphacorrection);
/*
Calculates the attenuation coefficient (alpha) of a given segment,
part of the wave number. A multiplicative factor alphacorrection is
also applied. Refer to Fletcher and Rossing (1998).
Parameters:
f: the frequency.
a: the radius of the tube in metres.
alphacorrection: the multiplicative attenuation coefficient
factor.
Returns:
The attenuation coefficient.
*/
complex charZ(double c, double rho, double a);
/*
Calculates the characteristic impedance of a given cylindrical pipe.
Refer to Fletcher and Rossing (1998).
Parameters:
c: the speed of sound.
rho: the density of air.
a: the radius of the tube in metres.
Returns:
The complex characteristic impedance of the tube.
*/
complex radiationZ(double f, double c, double rho, double a,
double flange);
/*
Calculates the radiation impedance of a termination.
Refer to Dalmont (2001) JSV 244, 505-34.
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
a: the radius of the tube in metres.
flange: -1 for a stopped termination, otherwise ratio of annulus
thickness to a.
Returns:
The complex radiation impedance.
*/
complex unflangedZ(double f, double c, double rho, double a);
/*
Calculates the radiation impedance of an unflanged pipe.
Refer to Dalmont (2001) JSV 244, 505-34.
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
a: the radius of the tube in metres.
Returns:

```

```

The complex radiation impedance.
*/
complex flangedZ(double f, double c, double rho, double a);
/*
Calculates the radiation impedance of a flanged pipe.
Refer to Dalmont (2001) JSV 244, 505-34.
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
a: the radius of the tube in metres.
Returns:
The complex radiation impedance.
*/
TransferMatrix tubeMatrix(double f, double c, double rho, double L,
    double a, double alphacorrection);
/*
Calculates the transfer matrix of a cylindrical tube.
The transfer matrix T relates pressure and flow at the input
to the pressure and flow at the output. Flow is positive
into the input and out of the output.
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
L: the length of the tube in metres.
a: the radius of the tube in metres.
alphacorrection: the multiplicative attenuation coefficient
factor.
Returns:
The transfer matrix of the tube.
*/
TransferMatrix coneMatrix(double f, double c, double rho, double L,
    double a1, double a2, double alphacorrection);
/*
Calculates the transfer matrix of a truncated cone.
The transfer matrix T relates pressure and flow at the input
to the pressure and flow at the output. Flow is positive
into the input and out of the output.
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
L: the length of the pipe in metres.
a1: the radius of the pipe at input in metres.
a2: the radius of the pipe at output in metres.
alphacorrection: the multiplicative attenuation coefficient
factor.
Returns:
The transfer matrix of the pipe.
*/
TransferMatrix discontinuityMatrix(double f, double c, double rho,
    double a1, double a2);
/*

```

Calculates the transfer matrix at a discontinuity.
Based on Pagneux et al. (1996) J. Acoust. Soc. Am. 100 p2034-48;
Parameters:
f: the frequency.
c: the speed of sound.
rho: the density of air.
a1: the radius at the input side of the discontinuity.
a2: the radius at the output side of the discontinuity.
Returns:
The transfer matrix of the discontinuity.

*/

#endif

Listing B.2: Acoustics.c

```

/*
Acoustics.c
By Paul Dickens, 2005

Acoustics library.
Refer to Acoustics.h for interface details.
*/

#include <math.h>
#include <gsl/gsl_sf_bessel.h>
#include "Acoustics.h"
#include "Complex.h"

double saturationVapourPressureWater(double T) {
    double C1, C2, C3, C4;
    C1 = 1.2811805e-5;
    C2 = 1.9509874e-2;
    C3 = 34.04926034;
    C4 = 6.3536311e3;
    return exp(C1*pow(T,2) - C2*T + C3 - C4/T);
}

double speedSound(double t, double p, double h, double x_c) {
    double f, T, p_sv, x_w;
    /* Calculate mole fraction of water vapour using equation in Cramer
       Appendix (p. 2515) */
    f = 1.00062 + 3.14e-8*p + 5.6e-7*pow(t,2);
    T = t + 273.15;
    p_sv = saturationVapourPressureWater(T);
    x_w = h*f*p_sv/p;
    return speedSoundCramer(t,p,x_w,x_c);
}

double speedSoundCramer(double t, double p, double x_w, double x_c) {
    int i;
    double c = 0;
    double a[] = {
        331.5024,
        0.603055,
        -0.000528,
        51.471935,
        0.1495874,
        -0.000782,
        -1.82e-7,
        3.73e-8,
        -2.93e-10,
        -85.20931,
        -0.228525,
        5.91e-5,
        -2.835149,
        -2.15e-13,
        29.179762,
        0.000486
    };
    double coeff[] = {

```

```

1.0,
t,
pow(t, 2),
x_w,
t * x_w,
pow(t, 2) * x_w,
p,
t * p,
pow(t, 2) * p,
x_c,
t * x_c,
pow(t, 2) * x_c,
pow(x_w, 2),
pow(p, 2),
pow(x_c, 2),
x_w * p * x_c
};

for (i = 0; i < 16; i++)
    c += a[i] * coeff[i];
return c;
}

double densityAirIdeal(double t, double p, double h, double x_c) {
    /* temperature in Kelvin */
    double T = 273.15 + t;
    /* gas constant for dry air in J/kg.K */
    double R_a = 287.05;
    /* gas constant for water vapour in J/kg.K */
    double R_w = 461.5;
    /* gas constant for carbon dioxide in J/kg.K */
    double R_c = 189;
    /* partial pressures of dry air, water vapour and CO2 */
    double p_a, p_w, p_c;

    p_w = h*saturationVapourPressureWater(T);
    p_c = x_c * p;
    p_a = p - p_w - p_c;

    return p_a / (R_a * T) + p_w / (R_w * T) + p_c / (R_c * T);
}

double densityAirGiacomo(double t, double p, double h, double x_c) {
    double T = 273.15 + t;
    double R = 8.314472;
    double M_w = 18.015e-3;
    double M_a, f, p_sv, x_w, Z;

    M_a = (28.9635 + 12.011 * (x_c - 0.0004)) * 1e-3;
    /* Calculate mole fraction of water vapour using equation in Cramer
    Appendix (p. 2515) */
    f = 1.00062 + 3.14e-8*p + 5.6e-7*pow(t,2);
    p_sv = saturationVapourPressureWater(T);
    x_w = h*f*p_sv/p;
    Z = compressibilityAir(t, p, x_w);
    return p * M_a * (1 - x_w * (1 - M_w / M_a)) / (Z * R * T);
}

double compressibilityAir(double t, double p, double x_w) {

```

```

double T = 273.15 + t;
double a_0, a_1, a_2, b_0, b_1, c_0, c_1, d, e;

a_0 = 1.62419e-6;
a_1 = -2.8969e-8;
a_2 = 1.0880e-10;
b_0 = 5.757e-6;
b_1 = -2.589e-8;
c_0 = 1.9297e-4;
c_1 = -2.285e-6;
d = 1.73e-11;
e = -1.034e-8;

return 1 - p * (a_0 + a_1 * t + a_2 * pow(t,2)
+ (b_0 + b_1 * t) * x_w + (c_0 + c_1 * t) * pow(x_w,2)) / T
+ pow(p,2) * (d + e * pow(x_w,2)) / pow(T,2);
}

complex waveNum(double f, double c, double a,
    double alphacorrection) {
/* implement wave number equation */
complex z;
double w = 2*M_PI*f;
z.Re = w/phaseVel(f, c, a);
z.Im = (-1)*attenCoeff(f, a, alphacorrection);
return z;
}

double phaseVel(double f, double c, double a) {
/* implement phase velocity equation */
return c*(1.0 - (1.65e-3/(a*sqrt(f))));
}

double attenCoeff(double f, double a, double alphacorrection) {
/* implement attenuation coefficient equation */
return alphacorrection*(3.0e-5*pow(f, 0.5))/a;
}

complex charZ(double c, double rho, double a) {
/* implement characteristic impedance equation */
double Zo = rho*c/(M_PI*a*a);
complex z = {Zo, 0};
return z;
}

complex radiationZ(double f, double c, double rho, double a,
    double flange) {
complex Z0, Z_u, Z_f, Z;
complex d_u, d_f, d;
double b, a_on_b;
double k = (2*M_PI*f)/c, ka = k * a;
double modR_edge, phaseR_edge;
complex R_noref, R_edge, R;

if (flange < 0.0) return inf;
if (flange == 0.0) return unflangedZ(f, c, rho, a);

/* calculate radiationZ for unflanged and flanged pipe */

```

```

Z_u = unflangedZ(f, c, rho, a);
Z_f = flangedZ(f, c, rho, a);

/* calculate the characteristic impedance */
Z0 = charZ(c, rho, a);

/* calculate complex end corrections for unflanged and flanged
   pipe */
d_u = divz(arctanz(divz(Z_u, multz(j, Z0))),real(k));
d_f = divz(arctanz(divz(Z_f, multz(j, Z0))),real(k));
b = a * (1+flange);
a_on_b = a / b;

/* calculate the length correction */
d = addz(addz(d_f,multz(real(a_on_b),subz(d_u,d_f))),
         real(0.057*a_on_b*(1 - pow(a_on_b,5))*a));

/* calculate the reflection coefficient (42) */
R_norefl = multz(real(-1), expz(multz(imaginary(-2 * k), d)));

modR_edge = -0.43 * (b - a) * a / pow(b,2)
           * pow(sin(k * b / (1.85 - a_on_b)),2);
phaseR_edge = -k * b * (1 + a_on_b * (2.3 - a_on_b - 0.3
           * pow(ka,2)));

R_edge = multz(real(modR_edge), expjz(real(phaseR_edge)));

R = addz(R_norefl,R_edge);

/* calculate and return the impedance */
Z = multz(Z0, divz(addz(one, R), subz(one, R)));
return Z;
}

complex unflangedZ(double f, double c, double rho, double a) {
    double k = (2*M_PI*f)/c;
    double ka = k*a;
    complex d;
    double modR;
    complex Z0 = charZ(c, rho, a), Z;

    /* define the end correction for the low frequency limit */
    d.Re = 0.6133 * a;
    /* determine the frequency-dependent end correction (14b) */
    d.Re = d.Re * ((1 + 0.044 * pow(ka,2)) / (1 + 0.19 * pow(ka,2))
                  - 0.02 * pow(sin(2 * ka),2));
    /* determine the modulus of the reflection coefficient (14c) */
    modR = (1 + 0.2 * ka - 0.084 * pow(ka,2)) / (1 + 0.2 * ka
          + (0.5 - 0.084) * pow(ka,2));
    /* calculate the imaginary part of the end correction */
    d.Im = log(modR) / (2 * k);
    /* calculate the impedance (9) */
    Z = multz(j, multz(Z0, tanz(multz(real(k), d))));
    return Z;
}

complex flangedZ(double f, double c, double rho, double a) {
    double k = (2*M_PI*f)/c;

```

```

double ka = k*a;
complex d;
double modR;
complex Z0 = charZ(c, rho, a), Z;

/* define the end correction for the low frequency limit */
d.Re = 0.8216 * a;
/* determine the frequency-dependent end correction (15a) */
d.Re = d.Re / (1 + pow(0.77 * ka,2) / (1 + 0.77 * ka));
/* determine the modulus of the reflection coefficient (15b) */
modR = (1 + 0.323 * ka - 0.077 * pow(ka,2)) / (1 + 0.323 * ka
+ (1 - 0.077) * pow(ka,2));
/* calculate the imaginary part of the end correction */
d.Im = log(modR) / (2 * k);
/* calculate the impedance (9) */
Z = multz(j, multz(Z0, tanz(multz(real(k), d))));
return Z;
}

TransferMatrix tubeMatrix(double f, double c, double rho, double L,
    double a, double alphacorrection) {
complex Zo, jkL, A, B, C, D;

/* check for zero length segment */
if(L == 0.0)
    return identity();

Zo = charZ(c, rho, a);
jkL = multz(j, multz(waveNum(f, c, a, alphacorrection), real(L)));

A = coshz(jkL);
B = multz(Zo, sinhZ(jkL));
C = divz(sinhZ(jkL), Zo);
D = A;
return createTransferMatrix(A, B, C, D);
}

TransferMatrix coneMatrix(double f, double c, double rho, double L,
    double a1, double a2, double alphacorrection) {
complex k, kL, kx1, kx2, theta1, theta2, sintheta1, sintheta2;
complex A, B, C, D;
double S1, S2, x1, x2, a;
double rhoc = rho*c;

/* check for zero length segment */
if(L == 0.0)
    return identity();

/* calculate apex distance for both ends of conical section
   based on similar triangles */
x1 = L/(a2/a1 - 1.0);

x2 = x1 + L;
a2 = a1*(1.0 + L/x1);

/* calculate areas of each end based on given radii */
S1 = M_PI*a1*a1;
S2 = M_PI*a2*a2;

```

```

/* use geometric mean of radii for attenuation purposes */
a = sqrt(a1*a2);

/* implement cone impedance calculation */
k = waveNum(f, c, a, alphacorrection);
kL = multz(k, real(L));
kx1 = multz(k, real(x1));
kx2 = multz(k, real(x2));
theta1 = arctanz(kx1);
theta2 = arctanz(kx2);
sintheta1 = sinz(theta1);
sintheta2 = sinz(theta2);

A = multz(real(-1), divz(sinz(subz(kL, theta2)), sintheta2));
B = multz(j, multz(real(rhoc/S2), sinz(kL)));
C = multz(imaginary(S1/rhoc), divz(sinz(addz(kL, subz(theta1,
    theta2))), multz(sintheta1, sintheta2)));
D = multz(real(S1/S2), divz(sinz(addz(kL, theta1)), sintheta1));
return createTransferMatrix(A, B, C, D);
}

TransferMatrix discontinuityMatrix(double f, double c, double rho,
    double a1, double a2) {
    TransferMatrix m = identitym();
    int n, N = 100;
    double F0n, gamma_n;
    double x = a1/a2;
    double omega = 2*M_PI*f;
    double S2 = M_PI*pow(a2,2);
    complex k_squared, k, Zchar, corr = zero;

    for (n = 1; n <= N; n++) {
        gamma_n = gsl_sf_bessel_zero_J1(n);
        F0n = 2*gsl_sf_bessel_J1(x * gamma_n)
            /(x * gamma_n * gsl_sf_bessel_J0(gamma_n));

        k_squared = real(pow(omega/c,2) - pow(gamma_n/a2,2));
        k = multz(real(-1), sqrtz(k_squared));

        Zchar = divz(real(omega*rho/S2), k);

        corr = addz(corr, multz(Zchar, real(pow(F0n,2))));
    }
    m->B = corr;
    return m;
}

```

Listing B.3: AnalyseNotes.c

```

/*
AnalyseNotes.c
By Andrew Botros, 2004
Modified by Paul Dickens, 2006

Analyses acoustic impedance spectra for playable notes.
*/

#include <stdio.h>
#include <string.h>
#include "Analysis.h"

int parseCommandLine(int argc, char** argv, int* displayharmonicity,
                     char** filename);

int main(int argc, char **argv) {
    int displayharmonicity;
    /* do not apply the pitch correction */
    int applypitchcorrection = 0;
    char* filename;
    /* check correct usage */
    if(!parseCommandLine(argc, argv, &displayharmonicity, &filename)) {
        fprintf(stderr,
                "Usage: AnalyseNotes [OPTIONS] <filename>\n\n");
        fprintf(stderr, " Options:\n");
        fprintf(stderr, "\t-h (Displays harmonicity data)\n\n");
        return -1;
    }
    /* perform analysis on impedance data file */
    Analysis(filename, applypitchcorrection, displayharmonicity, NOTES);
    return 0;
}

int parseCommandLine(int argc, char** argv, int* displayharmonicity,
                     char** filename) {
    int i;
    int hflag = 0;
    int numoptions = 1, numinputfiles = 1;
    int minargc = 1 + numinputfiles;
    int maxargc = minargc + numoptions;

    /* Check correct number of parameters */
    if((argc < minargc) || (argc > maxargc))
        return 0;

    /* Set default options */
    *displayharmonicity = 0;

    /* Check and set options */
    for(i = 1; i < (argc - numinputfiles); i++) {
        if(strcmp(argv[i], "-h") == 0) {
            if(hflag)
                return 0;
            *displayharmonicity = 1;
            hflag = 1;
        }
    }
}

```

```
    continue;
}
/* else invalid option */
fprintf(stderr, "Invalid option: %s\n", argv[i]);
return 0;
}

/* Set filename */
filename = argv[argc - numinputfiles];
return 1;
}
```

Listing B.4: Analysis.h

```

/*
Analysis.h
By Andrew Botros, 2001-2004
Modified by Paul Dickens, 2006

Analyses an acoustic impedance spectrum
for playable notes and multiphonics.
*/

```

```

#ifndef ANALYSIS_H_PROTECTOR
#define ANALYSIS_H_PROTECTOR

#include "Vector.h"
#include "Minima.h"

typedef enum {NOTES, MULTIPHONICS2, MULTIPHONICS3} AnalysisType;

void Analysis(char* filename, int applypitchcorrection,
              int displayharmonicity, AnalysisType at);
/*
Performs an analysis of the given impedance data file, sending
the required analysis results to stdout.

Parameters:
filename: the filename of the impedance data file
applypitchcorrection: boolean to flag the use of pitch correction
displayharmonicity: boolean to flag output of harmonicity data
at: the type of analysis (notes, two note multiphonics or
    three note multiphonics)
*/

```

```

int analyseNote(Minimum m, int applypitchcorrection,
                int displayharmonicity, int output);
/*
Determines the note and playability details of a Minimum if
playable, after correcting its pitch (output optional).

Parameters:
m: the Minimum to be analysed
applypitchcorrection: boolean to flag the use of pitch correction
displayharmonicity: boolean to flag output of harmonicity data
output: boolean to flag output

Returns:
1 if the Minimum is playable and within defined note range, 0 if
not.
*/

```

```

int playable(Minimum m);
/*
Determines whether a given Minimum is playable. Playability
is defined by the decision tree generated by C5.0. The decision
tree is implemented in this function.

Parameters:
m: the Minimum to be analysed

Returns:

```

```

    1 if playable, 0 if not.
*/
double playabilityLevel(Minimum m);
/*
Determines the playability level of a playable Minimum. The
playability level is determined by the rules generated by M5'.
The M5' rules are implemented in this function. In general,
3.0 is easily playable while 1.0 is difficult to play.
Parameters:
m: the Minimum to be analysed
Returns:
The playability level as a double.
*/
double strengthLevel(Minimum m);
/*
Determines the strength level of a playable Minimum. The
playability level is determined by the rules generated by M5'.
The M5' rules are implemented in this function. In general,
4.0 is strong and bright while 1.0 is weak and dark.
Parameters:
m: the Minimum to be analysed
Returns:
The strength level as a double.
*/
Minimum getCompletedMinimum(Minimum m);
/*
Creates a copy of a given Minimum with all empty fields initialised
to the average values of the expert set (cf M5').
Parameters:
m: the Minimum to be analysed
Returns:
A new Minimum with NaN fields replaced with expert set averages.
*/
void pitchCorrection(Minimum m);
/*
Corrects the pitch of a playable Minimum according to experimental
results from a number of flutists.
Parameters:
m: the Minimum to be analysed
*/
void analyseMultiphonics2(Vector playableminv);
/*
Determines if any two note multiphonics are playable in a set of
playable notes for a fingering. Multiphonics are defined as notes
which are "not harmonically related" (see code for details).
Outputs the multiphonic if so.
Parameters:
playableminv: the set of playable minima for a fingering
*/
void analyseMultiphonics3(Vector playableminv);
/*
Determines if any three note multiphonics are playable in a set of
*/

```

```

playable notes for a fingering. Multiphonics are defined as notes
which are "not harmonically related" (see code for details).
Doutputs the multiphonic if so.
Parameters:
    playableminv: the set of playable minima for a fingering
*/

int noteDistance(Minimum m1, Minimum m2, Vector playableminv);
/*
Calculates the distance in playable notes between two given minima.
Parameters:
    m1: the first minimum
    m2: the second minimum
    playableminv: the set of playable minima for a fingering
Returns:
    The number of playable minima between the given two minima, plus
    1.
    -1 if error.
*/

int pitchIndex(Vector minv);
/*
Calculates the sum of squares of cents of the given minima.
Parameters:
    minv: the set of minima
Returns:
    The result of the calculation (0 if no minima).
*/

char* allNotes(Vector playableminv);
/*
Concatenates the note strings of the playable notes for a
fingering, delimited by ';
Parameters:
    playableminv: the set of playable minima for a fingering
Returns:
    The concatenated null terminated string
*/

int harmonic(Minimum m1, Minimum m2, Vector playableminv);
/*
Determines whether two minima are harmonically related.
See code for definition of harmonic.
Parameters:
    m1: the first minimum
    m2: the second minimum
    playableminv: the set of playable minima for a fingering
Returns:
    1 if m1 and m2 are harmonic, 0 if not.
*/

int inArray(int* array, int size, int num);
/*
Determines whether an integer is present in a given sized
array of integers.
Parameters:
    array: the array of integers
    size: the number of elements in the array

```

```
num: the integer to find in the array
Returns:
  1 if the integer is present, 0 if not.
*/
#endif
```

Listing B.5: Analysis.c

```

printf("\n");
}
return;
}

int analyseNote(Minimum m, int applypitchcorrection,
    int displayharmonicity, int output) {
/* determine if Minimum is playable */
if(!playable(m))
    return 0;
/* determine playability and strength levels of playable Minimum */
double playability = playabilityLevel(m);
double strength = strengthLevel(m);

/* pitch correct Minimum if desired */
if(applypitchcorrection)
    pitchCorrection(m);
/* evaluate musical note from frequency (do not round) */
m->note = note(m->f, 0);
if(m->note == NULL)
    return 0;
else {
    if(output) {
        /* output notes */
        if(displayharmonicity)
            printf("%.1f\t%.1f\t%.1f\t%d\t%.1f\t%.0f\t%.1f",
                playability, strength, m->f, m->note->cents,
                m->Z, m->numharm, m->meanharmZ);
        else
            printf("%.1f\t%.1f\t%.1f\t%d\t%.1f",
                playability, strength, m->f, m->note->cents, m->Z);
    }
    return 1;
}
}

int playable(Minimum m) {
/* implement C5.0 decision tree */

if(m->Z <= 122.6) {
    if(!isValidNum(m->R_min_dZ) || (m->R_min_dZ <= -2.6))
        return 0;
    else {
        if(m->Z <= 116.7)
            return 1;
        else {
            if(!isValidNum(m->R_min_df) || (m->R_min_df <= 261.1))
                return 0;
            else
                return 1;
        }
    }
}
else
    return 0;
}

```

```

double playabilityLevel(Minimum m) {
    /* implement M5' model tree */

    Minimum mcomp = getCompletedMinimum(m);

    double LM, LM1, LM2;

    LM1 =
        - 0.0028 * mcomp->Z
        + 0.0090 * mcomp->numharm
        + 0.0023 * mcomp->meanharmZ
        + 0.0002 * mcomp->L_min_df
        + 0.0040 * mcomp->L_min_dZ
        - 0.0005 * mcomp->L_max_df
        + 0.0033 * mcomp->L_max_dZ
        + 0.0002 * mcomp->R_max_df
        + 2.7930;

    LM2 =
        - 0.0012 * mcomp->Z
        + 0.0975 * mcomp->numharm
        + 0.0210 * mcomp->meanharmZ
        + 0.0011 * mcomp->L_min_df
        + 0.0346 * mcomp->L_min_dZ
        - 0.0045 * mcomp->L_max_df
        + 0.0421 * mcomp->L_max_dZ
        + 0.0017 * mcomp->R_max_df
        - 1.8692;

    if(mcomp->Z <= 103.25)
        LM = LM1;
    else
        LM = LM2;

    return LM;
}

double strengthLevel(Minimum m) {
    /* implement M5' model tree */

    Minimum mcomp = getCompletedMinimum(m);

    double LM, LM1, LM2, LM3, LM4, LM5, LM6, LM7, LM8, LM9;

    LM1 =
        0.0001 * mcomp->f
        - 0.0059 * mcomp->Z
        + 0.1052 * mcomp->numharm
        + 0.0027 * mcomp->L_min_df
        + 0.0359 * mcomp->L_min_dZ
        + 0.0002 * mcomp->R_min_df
        - 0.0260 * mcomp->R_min_dZ
        - 0.0117 * mcomp->L_max_df
        + 0.0496 * mcomp->L_max_dZ
        + 0.0012 * mcomp->R_max_df
        - 0.0318 * mcomp->R_max_dZ
}

```

```

+ 3.4900;

LM2 =
 0.0014 * mcomp->f
- 0.0632 * mcomp->Z
+ 0.0397 * mcomp->numharm
- 0.0109 * mcomp->meanharmZ
+ 0.0020 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
+ 0.0001 * mcomp->R_min_dZ
+ 0.0021 * mcomp->L_max_df
- 0.0059 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 7.6356;

LM3 =
 0.0014 * mcomp->f
- 0.0632 * mcomp->Z
+ 0.0397 * mcomp->numharm
- 0.0109 * mcomp->meanharmZ
+ 0.0020 * mcomp->L_min_df
- 0.0001 * mcomp->R_min_df
+ 0.0001 * mcomp->R_min_dZ
+ 0.0021 * mcomp->L_max_df
- 0.0059 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 7.8619;

LM4 =
 0.0013 * mcomp->f
- 0.0632 * mcomp->Z
+ 0.0397 * mcomp->numharm
- 0.0109 * mcomp->meanharmZ
+ 0.0024 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
- 0.0403 * mcomp->R_min_dZ
- 0.0010 * mcomp->L_max_df
- 0.0059 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 8.8859;

LM5 =
 0.0013 * mcomp->f
- 0.0632 * mcomp->Z
+ 0.0397 * mcomp->numharm
- 0.0109 * mcomp->meanharmZ
+ 0.0040 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
+ 0.0127 * mcomp->R_min_dZ
- 0.0010 * mcomp->L_max_df
- 0.0059 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 8.1013;

LM6 =
 0.0012 * mcomp->f
- 0.0450 * mcomp->Z
+ 0.0149 * mcomp->numharm

```

```

- 0.0207 * mcomp->meanharmZ
+ 0.0007 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
- 0.0319 * mcomp->R_min_dZ
- 0.0028 * mcomp->L_max_df
+ 0.0122 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 9.4582;

LM7 =
0.0015 * mcomp->f
- 0.0450 * mcomp->Z
+ 0.0656 * mcomp->numharm
- 0.0290 * mcomp->meanharmZ
+ 0.0007 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
- 0.0140 * mcomp->R_min_dZ
- 0.0040 * mcomp->L_max_df
+ 0.0887 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 5.1755;

LM8 =
0.0015 * mcomp->f
- 0.0450 * mcomp->Z
+ 0.1310 * mcomp->numharm
- 0.0290 * mcomp->meanharmZ
+ 0.0007 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
- 0.0140 * mcomp->R_min_dZ
- 0.0040 * mcomp->L_max_df
+ 0.0887 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 4.8486;

LM9 =
- 0.0021 * mcomp->f
- 0.0450 * mcomp->Z
+ 0.0545 * mcomp->numharm
- 0.0290 * mcomp->meanharmZ
+ 0.0007 * mcomp->L_min_df
+ 0.0003 * mcomp->R_min_df
- 0.0140 * mcomp->R_min_dZ
- 0.0040 * mcomp->L_max_df
+ 0.0766 * mcomp->L_max_dZ
- 0.0003 * mcomp->R_max_df
+ 8.2141;

if(mcomp->L_max_df <= 139.55)
    LM = LM1;
else {
    if(mcomp->R_min_df <= 367.35) {
        if(mcomp->f <= 654.25) {
            if(mcomp->L_max_df <= 204.55)
                LM = LM2;
            else
                LM = LM3;
    }
}

```

```

    else {
        if(mcomp->R_min_dZ <= 10.65)
            LM = LM4;
        else
            LM = LM5;
    }
}
else {
    if(mcomp->L_max_df <= 213.9)
        LM = LM6;
    else {
        if(mcomp->L_max_dZ <= 54.6) {
            if(mcomp->numharm <= 3.5)
                LM = LM7;
            else
                LM = LM8;
        }
        else
            LM = LM9;
    }
}
}

return LM;
}

Minimum getCompletedMinimum(Minimum m) {
    Minimum mcomp = (Minimum)malloc(sizeof(*m));
    if(isValidNum(m->f)) mcomp->f = m->f; else mcomp->f = 1209.0;
    if(isValidNum(m->Z)) mcomp->Z = m->Z; else mcomp->Z = 108.0;
    if(isValidNum(m->B)) mcomp->B = m->B; else mcomp->B = 28.9;
    if(isValidNum(m->numharm)) mcomp->numharm = m->numharm;
    else mcomp->numharm = 2.6;
    if(isValidNum(m->meanharmZ)) mcomp->meanharmZ = m->meanharmZ;
    else mcomp->meanharmZ = 122.4;
    if(isValidNum(m->L_min_df)) mcomp->L_min_df = m->L_min_df;
    else mcomp->L_min_df = 329.0;
    if(isValidNum(m->L_min_dZ)) mcomp->L_min_dZ = m->L_min_dZ;
    else mcomp->L_min_dZ = -0.1;
    if(isValidNum(m->R_min_df)) mcomp->R_min_df = m->R_min_df;
    else mcomp->R_min_df = 357.2;
    if(isValidNum(m->R_min_dZ)) mcomp->R_min_dZ = m->R_min_dZ;
    else mcomp->R_min_dZ = 7.1;
    if(isValidNum(m->L_max_df)) mcomp->L_max_df = m->L_max_df;
    else mcomp->L_max_df = 129.7;
    if(isValidNum(m->L_max_dZ)) mcomp->L_max_dZ = m->L_max_dZ;
    else mcomp->L_max_dZ = 36.2;
    if(isValidNum(m->R_max_df)) mcomp->R_max_df = m->R_max_df;
    else mcomp->R_max_df = 259.8;
    if(isValidNum(m->R_max_dZ)) mcomp->R_max_dZ = m->R_max_dZ;
    else mcomp->R_max_dZ = 31.9;
    return mcomp;
}

/* legacy code - not used currently */
void pitchCorrection(Minimum m) {
    double f;
    double fcalc = m->f;

```

```

if(fcalc < 350.0)
    f = fcalc*pow(2.0, 13.0/1200.0)*pow(fcalc/233.0, 2.93/100.0);
if(fcalc >= 350.0 && fcalc < 1700.0)
    f = fcalc*pow(2.0, 50.0/1200.0)*pow(fcalc/233.0, -2.31/100.0);
if(fcalc >= 1700.0)
    f = fcalc*pow(2.0, -125.0/1200.0)*pow(fcalc/233.0, 2.6/100.0);
m->f = f;
}

void analyseMultiphonics2(Vector playableminv) {
    int i, j;
    Minimum m1, m2;
    Vector minv;
    /* for each possible pair of playable notes */
    for(i = 0; i < sizeVector(playableminv) - 1; i++) {
        m1 = (Minimum)elementAt(playableminv, i);
        for(j = i + 1; j < sizeVector(playableminv); j++) {
            m2 = (Minimum)elementAt(playableminv, j);
            /* if the notes are not harmonic, output */
            if(!harmonic(m1, m2, playableminv)) {
                minv = createVector();
                addElement(minv, m1);
                addElement(minv, m2);
                printf("%s\t%d\t%s\t%d\t%d\t%d\t%s\n",
                    m1->note->name, m1->note->midi,
                    m2->note->name, m2->note->midi,
                    noteDistance(m1, m2, playableminv), pitchIndex(minv),
                    allNotes(playableminv));
            }
        }
    }
}

void analyseMultiphonics3(Vector playableminv) {
    int i, j, k;
    Minimum m1, m2, m3;
    Vector minv;
    /* for each possible trio of playable notes */
    for(i = 0; i < sizeVector(playableminv) - 2; i++) {
        m1 = (Minimum)elementAt(playableminv, i);
        for(j = i + 1; j < sizeVector(playableminv) - 1; j++) {
            m2 = (Minimum)elementAt(playableminv, j);
            for(k = j + 1; k < sizeVector(playableminv); k++) {
                m3 = (Minimum)elementAt(playableminv, k);
                /* if all pairs of the trio are not harmonic, output */
                if(!harmonic(m1, m2, playableminv) && !harmonic(m2, m3,
                    playableminv)
                    && !harmonic(m1, m3, playableminv)) {
                    minv = createVector();
                    addElement(minv, m1);
                    addElement(minv, m2);
                    addElement(minv, m3);
                    printf("%s\t%d\t%s\t%d\t%d\t%d\t%d\t%s\n",
                        m1->note->name, m1->note->midi,
                        m2->note->name, m2->note->midi,
                        m3->note->name, m3->note->midi,
                        noteDistance(m1, m3, playableminv), pitchIndex(minv),
                        allNotes(playableminv));
                }
            }
        }
    }
}

```

```

        }
    }
}

int noteDistance(Minimum m1, Minimum m2, Vector playableminv) {
    int i;
    Minimum m;
    int posm1 = -1, posm2 = -1;
    /* find the positions of each minimum in vector */
    for(i = 0; i < sizeVector(playableminv); i++) {
        m = (Minimum)elementAt(playableminv, i);
        if(m == m1)
            posm1 = i;
        if(m == m2)
            posm2 = i;
    }
    /* check for an invalid condition */
    if((posm1 == -1) || (posm2 == -1) || (posm1 >= posm2))
        return -1;
    else
        return (posm2 - posm1);
}

int pitchIndex(Vector minv) {
    int i;
    Minimum m;
    int index = 0;
    /* calculate sum of squares of cents */
    for(i = 0; i < sizeVector(minv); i++) {
        m = (Minimum)elementAt(minv, i);
        index = index + (m->note->cents)*(m->note->cents);
    }
    return index;
}

char* allNotes(Vector playableminv) {
    int i;
    Minimum m;
    char* all_notes = (char*)malloc(BUFSIZ*sizeof(char));
    /* for each note that is playable, concatenate into
       one string delimited by a ';' */
    for(i = 0; i < sizeVector(playableminv); i++) {
        m = (Minimum)elementAt(playableminv, i);
        if(i == 0)
            sprintf(all_notes, noteString(m->note));
        else {
            strcat(all_notes, ",");
            strcat(all_notes, noteString(m->note));
        }
    }
    /* return concatenated string */
    return all_notes;
}

int harmonic(Minimum m1, Minimum m2, Vector playableminv) {
    int i;

```

```

Minimum m;
int n = sizeVector(playableminv);
int midis[n];
int dmidi;
int num_harmonic_midis = 11;
int harmonic_midis[] = {0, 12, 19, 24, 28, 31, 33, 34, 36, 38, 40};

/* get the midi numbers of all playable minima. The difference in
   midi number of two notes is the number of semitones between
   them. */
for(i = 0; i < n; i++) {
    m = (Minimum)elementAt(playableminv, i);
    midis[i] = m->note->midi;
}

/* calculate number of semitones between two minima */
dmidi = m2->note->midi - m1->note->midi;

/* if the minima are directly harmonically related (integer
   frequency ratio) */
if(inArray(harmonic_midis, num_harmonic_midis, dmidi))
    return 1;

/* if the minima have a common playable fundamental */
if((dmidi == 7 || dmidi == 16 || dmidi == 21 || dmidi == 22 || dmidi
== 26)
   && inArray(midis, n, m1->note->midi - 12))
    return 1;
if((dmidi == 5 || dmidi == 9 || dmidi == 14 || dmidi == 15 || dmidi
== 17 || dmidi == 21)
   && inArray(midis, n, m1->note->midi - 19))
    return 1;
if((dmidi == 4 || dmidi == 7 || dmidi == 9 || dmidi == 10 || dmidi
== 14 || dmidi == 16)
   && inArray(midis, n, m1->note->midi - 24))
    return 1;
if((dmidi == 3 || dmidi == 5 || dmidi == 6 || dmidi == 8 || dmidi ==
10)
   && inArray(midis, n, m1->note->midi - 28))
    return 1;
if((dmidi == 2 || dmidi == 3 || dmidi == 5 || dmidi == 7 || dmidi ==
9)
   && inArray(midis, n, m1->note->midi - 31))
    return 1;
if((dmidi == 3 || dmidi == 5 || dmidi == 7)
   && inArray(midis, n, m1->note->midi - 33))
    return 1;
if((dmidi == 2 || dmidi == 4 || dmidi == 6)
   && inArray(midis, n, m1->note->midi - 34))
    return 1;
if((dmidi == 2 || dmidi == 4)
   && inArray(midis, n, m1->note->midi - 36))
    return 1;
if(dmidi == 2 && inArray(midis, n, m1->note->midi - 38))
    return 1;

/* otherwise, not harmonic */
return 0;

```

```
}
```

```
int inArray(int* array, int size, int num) {
    int i;
    /* look for num in array */
    for(i = 0; i < size; i++) {
        if(array[i] == num)
            return 1;
    }
    return 0;
}
```

Listing B.6: BbClarinet.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE woodwind SYSTEM "woodwind.dtd">
<woodwind description="Clarinet Bb Yamaha CX1">
    <!-- Note this file includes all geometry beginning from
        50 mm into the mouthpiece from the end of the tenon.
        Also note that the keys have not been measured for this
        instrument and so no keys are included in this model. -->
    <downstream flange="0.0">
        <bore>
            <radius1>6.9</radius1>
            <radius2>6.95</radius2>
            <length>5.0</length>
        </bore>
        <bore>
            <radius1>6.95</radius1>
            <radius2>7.0</radius2>
            <length>2.0</length>
        </bore>
        <bore>
            <radius1>7.0</radius1>
            <radius2>7.15</radius2>
            <length>10.0</length>
        </bore>
        <bore>
            <radius1>7.15</radius1>
            <radius2>7.2</radius2>
            <length>5.0</length>
        </bore>
        <bore>
            <radius1>7.2</radius1>
            <radius2>7.25</radius2>
            <length>4.0</length>
        </bore>
        <bore>
            <radius1>7.25</radius1>
            <radius2>7.3</radius2>
            <length>2.0</length>
        </bore>
        <bore>
            <radius1>7.3</radius1>
            <radius2>7.35</radius2>
            <length>3.0</length>
        </bore>
        <bore>
            <radius1>7.35</radius1>
            <radius2>7.4</radius2>
            <length>2.0</length>
        </bore>
        <bore>
            <radius1>7.4</radius1>
            <radius2>7.45</radius2>
            <length>11.0</length>
        </bore>
        <bore>
            <radius1>7.45</radius1>
            <radius2>7.5</radius2>
            <length>6.0</length>
        </bore>
    </downstream>
</woodwind>
```

```
</bore>
<bore>
  <radius1>7.5</radius1>
  <radius2>7.45</radius2>
  <length>10.8</length>
</bore>
<bore>
  <radius1>7.45</radius1>
  <radius2>7.4</radius2>
  <length>10.0</length>
</bore>
<bore>
  <radius1>7.4</radius1>
  <radius2>7.4</radius2>
  <length>8.1</length>
</bore>
<bore>
  <radius1>7.5</radius1>
  <radius2>7.45</radius2>
  <length>20.0</length>
</bore>
<bore>
  <radius1>7.45</radius1>
  <radius2>7.45</radius2>
  <length>11.5</length>
</bore>
<bore>
  <radius1>7.45</radius1>
  <radius2>7.44298245614035</radius2>
  <length>5.6</length>
</bore>
<hole>
  <radius>1.4</radius>
  <length>12.6</length>
  <boreradius>7.44298245614035</boreradius>
</hole>
<bore>
  <radius1>7.44298245614035</radius1>
  <radius2>7.4293233082706776</radius2>
  <length>10.9</length>
</bore>
<hole>
  <radius>2.4</radius>
  <length>6.8</length>
  <boreradius>7.4293233082706776</boreradius>
</hole>
<bore>
  <radius1>7.4293233082706776</radius1>
  <radius2>7.396115288220552</radius2>
  <length>26.5</length>
</bore>
<hole>
  <radius>2.8</radius>
  <length>6.7</length>
  <boreradius>7.396115288220552</boreradius>
</hole>
<bore>
  <radius1>7.396115288220552</radius1>
```

```
<radius2>7.3845864661654135</radius2>
<length>9.2</length>
</bore>
<hole>
<radius>2.6</radius>
<length>6.8</length>
<boreradius>7.3845864661654135</boreradius>
</hole>
<bore>
<radius1>7.3845864661654135</radius1>
<radius2>7.371303258145364</radius2>
<length>10.6</length>
</bore>
<hole>
<radius>2.6</radius>
<length>6.6</length>
<boreradius>7.371303258145364</boreradius>
</hole>
<bore>
<radius1>7.371303258145364</radius1>
<radius2>7.35</radius2>
<length>17.0</length>
</bore>
<hole>
<radius>2.4</radius>
<length>5.1</length>
<boreradius>7.35</boreradius>
</hole>
<bore>
<radius1>7.35</radius1>
<radius2>7.35</radius2>
<length>8.7</length>
</bore>
<hole>
<radius>3.7</radius>
<length>10.8</length>
<boreradius>7.35</boreradius>
</hole>
<bore>
<radius1>7.35</radius1>
<radius2>7.35</radius2>
<length>2.0</length>
</bore>
<hole>
<radius>2.3</radius>
<length>6.5</length>
<boreradius>7.35</boreradius>
</hole>
<bore>
<radius1>7.35</radius1>
<radius2>7.35</radius2>
<length>11.1</length>
</bore>
<hole>
<radius>2.6</radius>
<length>9.1</length>
<boreradius>7.35</boreradius>
</hole>
```

```
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>18.9</length>
</bore>
<hole>
  <radius>2.2</radius>
  <length>6.9</length>
  <boreradius>7.35</boreradius>
</hole>
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>13.5</length>
</bore>
<hole>
  <radius>3.3</radius>
  <length>9.2</length>
  <boreradius>7.35</boreradius>
</hole>
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>2.5</length>
</bore>
<hole>
  <radius>2.6</radius>
  <length>6.7</length>
  <boreradius>7.35</boreradius>
</hole>
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>2.3</length>
</bore>
<hole>
  <radius>2.7</radius>
  <length>6.4</length>
  <boreradius>7.35</boreradius>
</hole>
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>18.7</length>
</bore>
<hole>
  <radius>3.7</radius>
  <length>7.3</length>
  <boreradius>7.35</boreradius>
</hole>
<bore>
  <radius1>7.35</radius1>
  <radius2>7.35</radius2>
  <length>10.0</length>
</bore>
<hole>
  <radius>2.4</radius>
  <length>6.4</length>
```

```
<boreradius>7.35</boreradius>
</hole>
<bore>
<radius1>7.35</radius1>
<radius2>7.3</radius2>
<length>23.6</length>
</bore>
<bore>
<radius1>7.25</radius1>
<radius2>7.253079884504331</radius2>
<length>6.4</length>
</bore>
<hole>
<radius>3.5</radius>
<length>6.7</length>
<boreradius>7.253079884504331</boreradius>
</hole>
<bore>
<radius1>7.253079884504331</radius1>
<radius2>7.260490856592877</radius2>
<length>15.4</length>
</bore>
<hole>
<radius>3.9</radius>
<length>8.9</length>
<boreradius>7.260490856592877</boreradius>
</hole>
<bore>
<radius1>7.260490856592877</radius1>
<radius2>7.262945139557266</radius2>
<length>5.1</length>
</bore>
<hole>
<radius>3.8</radius>
<length>6.3</length>
<boreradius>7.262945139557266</boreradius>
</hole>
<bore>
<radius1>7.262945139557266</radius1>
<radius2>7.273195380173243</radius2>
<length>21.3</length>
</bore>
<hole>
<radius>3.8</radius>
<length>8.8</length>
<boreradius>7.273195380173243</boreradius>
</hole>
<bore>
<radius1>7.273195380173243</radius1>
<radius2>7.284119345524543</radius2>
<length>22.7</length>
</bore>
<hole>
<radius>4.7</radius>
<length>8.8</length>
<boreradius>7.284119345524543</boreradius>
</hole>
<bore>
```

```
<radius1>7.284119345524543</radius1>
<radius2>7.2996150144369585</radius2>
<length>32.2</length>
</bore>
<hole>
  <radius>5.3</radius>
  <length>5.7</length>
  <boreradius>7.2996150144369585</boreradius>
</hole>
<bore>
  <radius1>7.2996150144369585</radius1>
  <radius2>7.3</radius2>
  <length>0.8</length>
</bore>
<bore>
  <radius1>7.3</radius1>
  <radius2>7.4529069767441865</radius2>
  <length>26.3</length>
</bore>
<hole>
  <radius>6.0</radius>
  <length>5.2</length>
  <boreradius>7.4529069767441865</boreradius>
</hole>
<bore>
  <radius1>7.4529069767441865</radius1>
  <radius2>7.5</radius2>
  <length>8.1</length>
</bore>
<bore>
  <radius1>7.5</radius1>
  <radius2>7.75</radius2>
  <length>17.2</length>
</bore>
<bore>
  <radius1>7.75</radius1>
  <radius2>7.892307692307692</radius2>
  <length>7.4</length>
</bore>
<hole>
  <radius>5.7</radius>
  <length>5.0</length>
  <boreradius>7.892307692307692</boreradius>
</hole>
<bore>
  <radius1>7.892307692307692</radius1>
  <radius2>8.0</radius2>
  <length>5.6</length>
</bore>
<bore>
  <radius1>8.0</radius1>
  <radius2>8.25</radius2>
  <length>8.0</length>
</bore>
<bore>
  <radius1>8.25</radius1>
  <radius2>8.5</radius2>
  <length>10.0</length>
```

```
</bore>
<bore>
  <radius1>8.5</radius1>
  <radius2>8.75</radius2>
  <length>7.0</length>
</bore>
<bore>
  <radius1>8.75</radius1>
  <radius2>8.98125</radius2>
  <length>7.4</length>
</bore>
<hole>
  <radius>5.4</radius>
  <length>4.6</length>
  <boreradius>8.98125</boreradius>
</hole>
<bore>
  <radius1>8.98125</radius1>
  <radius2>9.0</radius2>
  <length>0.6</length>
</bore>
<bore>
  <radius1>9.0</radius1>
  <radius2>9.25</radius2>
  <length>7.0</length>
</bore>
<bore>
  <radius1>9.25</radius1>
  <radius2>9.5</radius2>
  <length>6.4</length>
</bore>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.75</radius2>
  <length>6.8</length>
</bore>
<bore>
  <radius1>9.75</radius1>
  <radius2>10.0</radius2>
  <length>4.5</length>
</bore>
<bore>
  <radius1>10.0</radius1>
  <radius2>10.25</radius2>
  <length>3.7</length>
</bore>
<bore>
  <radius1>10.25</radius1>
  <radius2>10.5</radius2>
  <length>3.0</length>
</bore>
<bore>
  <radius1>10.5</radius1>
  <radius2>10.75</radius2>
  <length>3.0</length>
</bore>
<bore>
  <radius1>10.75</radius1>
```

```
<radius2>11.0</radius2>
<length>3.6</length>
</bore>
<bore>
<radius1>11.15</radius1>
<radius2>12.5</radius2>
<length>12.8</length>
</bore>
<bore>
<radius1>12.5</radius1>
<radius2>15.0</radius2>
<length>16.0</length>
</bore>
<bore>
<radius1>15.0</radius1>
<radius2>16.25</radius2>
<length>7.0</length>
</bore>
<bore>
<radius1>16.25</radius1>
<radius2>19.0</radius2>
<length>14.0</length>
</bore>
<bore>
<radius1>19.0</radius1>
<radius2>25.0</radius2>
<length>25.0</length>
</bore>
<bore>
<radius1>25.0</radius1>
<radius2>29.5</radius2>
<length>12.0</length>
</bore>
</downstream>
</woodwind>
```

Listing B.7: ClassicalFlute.xml

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE woodwind SYSTEM "woodwind.dtd">
<woodwind description="McGee classical flute">
  <embouchurehole>
    <radiusin>5.7</radiusin>
    <radiusout>5.5</radiusout>
    <length>3.5</length>
    <boreradius>9.5</boreradius>
  </embouchurehole>
  <upstream flange="-1.0">
    <bore>
      <radius1>9.5</radius1>
      <radius2>9.5</radius2>
      <length>19.0</length>
    </bore>
  </upstream>
  <downstream flange="0.75">
    <bore>
      <radius1>9.5</radius1>
      <radius2>9.5</radius2>
      <length>139.5</length>
    </bore>
    <bore>
      <radius1>10.0</radius1>
      <radius2>10.0</radius2>
      <length>12.8</length>
    </bore>
    <bore>
      <radius1>9.5</radius1>
      <radius2>9.2</radius2>
      <length>2.0</length>
    </bore>
    <bore>
      <radius1>9.2</radius1>
      <radius2>8.6</radius2>
      <length>83.5</length>
    </bore>
    <hole name="L1">
      <radius>4.0</radius>
      <length>4.8</length>
      <boreradius>8.6</boreradius>
    </hole>
    <bore>
      <radius1>8.6</radius1>
      <radius2>8.412073490813647</radius2>
      <length>17.9</length>
    </bore>
    <hole name="C">
      <radius>3.2</radius>
      <length>4.5</length>
      <boreradius>8.412073490813647</boreradius>
      <key>
        <radius>6.7</radius>
        <holeradius>0.0</holeradius>
        <height>4.0</height>
        <thickness>4.0</thickness>
        <wallThickness>3.0</wallThickness>
      </key>
    </hole>
  </downstream>
</woodwind>

```

```
<chimneyHeight>0.0</chimneyHeight>
</key>
</hole>
<bore>
<radius1>8.412073490813647</radius1>
<radius2>8.2</radius2>
<length>20.2</length>
</bore>
<hole name="L2">
<radius>5.1</radius>
<length>4.9</length>
<boreradius>8.2</boreradius>
</hole>
<bore>
<radius1>8.2</radius1>
<radius2>8.032743362831859</radius2>
<length>18.9</length>
</bore>
<hole name="Bb">
<radius>3.0</radius>
<length>4.5</length>
<boreradius>8.032743362831859</boreradius>
<key>
<radius>5.7</radius>
<holeradius>0.0</holeradius>
<height>2.0</height>
<thickness>2.5</thickness>
<wallThickness>3.0</wallThickness>
<chimneyHeight>0.0</chimneyHeight>
</key>
</hole>
<bore>
<radius1>8.032743362831859</radius1>
<radius2>7.9</radius2>
<length>15.0</length>
</bore>
<hole name="L3">
<radius>4.0</radius>
<length>5.0</length>
<boreradius>7.9</boreradius>
</hole>
<bore>
<radius1>7.9</radius1>
<radius2>7.665720081135903</radius2>
<length>23.1</length>
</bore>
<hole name="G#">
<radius>3.0</radius>
<length>4.5</length>
<boreradius>7.665720081135903</boreradius>
<key>
<radius>6.0</radius>
<holeradius>0.0</holeradius>
<height>2.7</height>
<thickness>3.0</thickness>
<wallThickness>3.0</wallThickness>
<chimneyHeight>0.0</chimneyHeight>
</key>
```

```
</hole>
<bore>
  <radius1>7.665720081135903</radius1>
  <radius2>7.4</radius2>
  <length>26.2</length>
</bore>
<bore>
  <radius1>7.5</radius1>
  <radius2>7.4</radius2>
  <length>8.6</length>
</bore>
<hole name="R1">
  <radius>4.8</radius>
  <length>5.1</length>
  <boreradius>7.4</boreradius>
</hole>
<bore>
  <radius1>7.4</radius1>
  <radius2>7.1</radius2>
  <length>31.0</length>
</bore>
<hole name="R2">
  <radius>5.5</radius>
  <length>5.0</length>
  <boreradius>7.1</boreradius>
</hole>
<bore>
  <radius1>7.1</radius1>
  <radius2>6.8</radius2>
  <length>20.4</length>
</bore>
<hole name="F">
  <radius>4.0</radius>
  <length>4.5</length>
  <boreradius>6.8</boreradius>
  <key>
    <radius>6.7</radius>
    <holeradius>0.0</holeradius>
    <height>1.9</height>
    <thickness>4.0</thickness>
    <wallThickness>3.0</wallThickness>
    <chimneyHeight>0.0</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>6.8</radius1>
  <radius2>6.8</radius2>
  <length>15.5</length>
</bore>
<hole name="R3">
  <radius>3.3</radius>
  <length>5.2</length>
  <boreradius>6.8</boreradius>
</hole>
<bore>
  <radius1>6.8</radius1>
  <radius2>6.6</radius2>
  <length>35.4</length>
```

```
</bore>
<bore>
  <radius1>6.8</radius1>
  <radius2>6.8</radius2>
  <length>52.3</length>
</bore>
<hole name="C#">
  <radius>6.5</radius>
  <length>5.1</length>
  <boreradius>6.8</boreradius>
</hole>
<bore>
  <radius1>6.8</radius1>
  <radius2>6.7</radius2>
  <length>36.7</length>
</bore>
<hole name="C">
  <radius>5.2</radius>
  <length>5.0</length>
  <boreradius>6.7</boreradius>
</hole>
<bore>
  <radius1>6.7</radius1>
  <radius2>6.7</radius2>
  <length>39.2</length>
</bore>
</downstream>
</woodwind>
```

Listing B.8: Complex.h

```

/*
Complex.h
By Andrew Botros, 2000-2004

Hyperbolic trig fns and expz added by Paul Dickens, 2005

Complex arithmetic library.
*/

```

```

#ifndef COMPLEX_H_PROTECTOR
#define COMPLEX_H_PROTECTOR

/* complex: { Re(z), Im(z) } */
typedef struct complex_str {
    double Re;
    double Im;
} complex;

/* externally defined complex numbers zero, one, j and inf for
convenience */
extern complex zero;
extern complex one;
extern complex j;
extern complex inf;

double modz(complex z);
/*
Parameters:
z: given complex number.
Returns:
|z| ... Modulus of z
*/

double argz(complex z);
/*
Parameters:
z: given complex number.
Returns:
arg z ... Principal arg of z
*/

int equalz(complex z1, complex z2);
/*
Parameters:
z1: given complex number.
z2: given complex number.
Returns:
1 if z1 equals z2, 0 otherwise.
*/

complex real(double x);
/*
Parameters:
x: given double.

```

```

Returns:
  A complex struct z with Re z = x and Im z = 0
*/
complex imaginary(double y);
/*
Parameters:
  y: given double
Returns:
  A complex struct z with Re z = 0 and Im z = y
*/
complex addz(complex z1, complex z2);
/*
Parameters:
  z1: given complex number.
  z2: given complex number.
Returns:
  z1 + z2 ... The sum of z1 and z2
*/
complex subz(complex z1, complex z2);
/*
Parameters:
  z1: given complex number.
  z2: given complex number.
Returns:
  z1 - z2 ... z2 subtracted from z1
*/
complex multz(complex z1, complex z2);
/*
Parameters:
  z1: given complex number.
  z2: given complex number.
Returns:
  z1 x z2 ... The product of z1 and z2
*/
complex divz(complex z1, complex z2);
/*
Parameters:
  z1: given complex number.
  z2: given complex number.
Returns:
  z1 / z2 ... z1 divided by z2
*/
complex expz(complex z1);
/*
Parameters:
  z1: given complex number.
Returns:
  exp(z1) ... The complex exponential of z1
*/
complex expjz(complex z1);
/*

```

```

Parameters:
z1: given complex number.

Returns:
exp(j x z1) ... The complex exponential of (j x z1)
*/
complex coshz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
cosh z1 ... The complex hyperbolic cosine of z1
*/
complex sinhz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
sinh z1 ... The complex hyperbolic sine of z1
*/
complex cosz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
cos z1 ... The complex cosine of z1
*/
complex sinz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
sin z1 ... The complex sine of z1
*/
complex tanz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
tan z1 ... The complex tan of z1
*/
complex cotz(complex z1);
/*
Parameters:
z1: given complex number.

Returns:
cot z1 ... The complex cotan of z1
*/
complex logz(complex z1);
/*
Parameters:
z1: given complex number.

```

```
>Returns:  
    log z1 ... The complex principal logarithm of z1  
*/  
  
complex arctanz(complex z1);  
/*  
Parameters:  
    z1: given complex number.  
Returns:  
    arctan z1 ... The complex inverse tan of z1  
*/  
  
complex parallel(complex z1, complex z2);  
/*  
Parameters:  
    z1: given complex number.  
    z2: given complex number.  
Returns:  
    The complex representation of z1 // z2 (in the electrical sense).  
*/  
  
complex sqrtz(complex z1);  
/*  
Parameters:  
    z1: given complex number.  
Returns:  
    The square root of z1.  
*/  
  
void printComplex(complex z);  
/*  
Prints a complex struct to stdout.  
Parameters:  
    z: given complex number.  
*/  
  
#endif
```

Listing B.9: Complex.c

```

/*
Complex.c
By Andrew Botros, 2000-2004

Hyperbolic trig fns and expz added by Paul Dickens, 2005

Complex arithmetic library.
Refer to Complex.h for interface details.

Refer to the following for algebraic expressions:
Brown, J.W., Churchill, R.W., 1996.
Complex Variables and Applications.
6th ed. McGraw-Hill, New York,
*/

```

```

#include <stdio.h>
#include <math.h>
#include <float.h>
#include "Complex.h"

/* zero, one, j and inf defined for convenience */
complex zero = {0.0, 0.0};
complex j = {0.0, 1.0};
complex one = {1.0, 0.0};
complex inf = {DBL_MAX, 0.0};

double modz(complex z) {
    return sqrt(z.Re*z.Re + z.Im*z.Im);
}

double argz(complex z) {
    return atan2(z.Im, z.Re);
}

complex real(double x) {
    complex z = {x, 0.0};
    return z;
}

complex imaginary(double y) {
    complex z = {0.0, y};
    return z;
}

int equalz(complex z1, complex z2) {
    return (z1.Re == z2.Re) && (z1.Im == z2.Im);
}

complex addz(complex z1, complex z2) {
    complex z;
    z.Re = z1.Re + z2.Re;
    z.Im = z1.Im + z2.Im;
    return z;
}

```

```

complex subz(complex z1, complex z2) {
    complex z;
    z.Re = z1.Re - z2.Re;
    z.Im = z1.Im - z2.Im;
    return z;
}

complex multz(complex z1, complex z2) {
    complex z;
    z.Re = z1.Re*z2.Re - z1.Im*z2.Im;
    z.Im = z1.Re*z2.Im + z1.Im*z2.Re;
    return z;
}

complex divz(complex z1, complex z2) {
    if(equalz(z2, zero)) return inf;
    complex z;
    z.Re = (z1.Re*z2.Re + z1.Im*z2.Im)/(z2.Re*z2.Re + z2.Im*z2.Im);
    z.Im = (z1.Im*z2.Re - z1.Re*z2.Im)/(z2.Re*z2.Re + z2.Im*z2.Im);
    return z;
}

complex expz(complex z1) {
    complex z;
    z.Re = exp(z1.Re)*cos(z1.Im);
    z.Im = exp(z1.Re)*sin(z1.Im);
    return z;
}

complex expjz(complex z1) {
    return expz(multz(j, z1));
}

complex coshz(complex z1) {
    complex minus_one = {-1.0, 0.0};
    complex two = {2.0, 0.0};
    complex minus_z1 = multz(minus_one, z1);
    return divz(addz(expz(z1), expz(minus_z1)), two);
}

complex sinhz(complex z1) {
    complex minus_one = {-1.0, 0.0};
    complex two = {2.0, 0.0};
    complex minus_z1 = multz(minus_one, z1);
    return divz(subz(expz(z1), expz(minus_z1)), two);
}

complex cosz(complex z1) {
    return coshz(multz(j, z1));
}

complex sinz(complex z1) {
    return divz(sinhz(multz(j, z1)), j);
}

complex tanz(complex z1) {

```

```
    return divz(sinz(z1), cosz(z1));
}

complex cotz(complex z1) {
    return divz(cosz(z1), sinz(z1));
}

complex logz(complex z1) {
    complex result;
    result.Re = log(modz(z1));
    result.Im = argz(z1);
    return result;
}

complex arctanz(complex z1) {
    complex two = {2.0, 0.0};
    return multz(divz(j, two), logz(divz(addz(j, z1), subz(j, z1))));
}

complex parallel(complex z1, complex z2) {
    return divz(multz(z1, z2), addz(z1, z2));
}

complex sqrtz(complex z1) {
    double A = modz(z1);
    double theta = argz(z1);
    A = sqrt(A);
    theta = theta / 2;
    return multz(real(A), expjz(real(theta)));
}

void printComplex(complex z) {
    printf("%e%+ei", z.Re, z.Im);
}
```

Listing B.10: Impedance.c

```
/*
Impedance.c
By Paul Dickens, 2005

Physical model of the acoustic impedance of a flute.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "Woodwind.h"
#include "ParseXML.h"

int parseCommandLine(int argc, char** argv, char** holestring,
                     double* temp, double* humid, double* flo, double* fhi,
                     double* fres, double* entryratio, char** xml_filename);

/* Default parameter values */
#define TEMP 25.0
#define HUMID 0.5
#define X_CO2 0.0004
#define FLO 200.0
#define FHI 4000.0
#define FRES 2.0
#define ENTRYRATIO 1.0

int main(int argc, char** argv) {
    char* holestring;
    double temp, humid;
    double f, flo, fhi, fres, entryratio;
    char* xml_filename;
    Woodwind instrument;
    complex Z;

    /* check correct usage */
    if(!parseCommandLine(argc, argv, &holestring, &temp, &humid,
                         &flo, &fhi, &fres, &entryratio, &xml_filename)) {
        fprintf(stderr, "Usage: Impedance [OPTIONS] <XML file>\n\n");
        fprintf(stderr, " Options:\n");
        fprintf(stderr, "\t-s <holestring>\n");
        fprintf(stderr, "\t-t <temperature> (default 25.0 degC)\n");
        fprintf(stderr, "\t-u <humidity> (default 0.5)\n");
        fprintf(stderr, "\t-l <flo> (default 100.0)\n");
        fprintf(stderr, "\t-h <fhi> (default 4000.0)\n");
        fprintf(stderr, "\t-r <fres> (default 2.0)\n");
        fprintf(stderr, "\t-e <entryratio> (default 1.0)\n\n");
        fprintf(stderr, " <holestring>:\n");
        fprintf(stderr,
                "\t- Optional if no holes are defined in XML file.\n");
        fprintf(stderr, "\t- Must be a sequence of '0' (open hole) ");
        fprintf(stderr, "and 'X' (closed hole) characters.\n");
        fprintf(stderr,

```

```

    "\t- Length must be equal to the number of defined holes.\n");
    fprintf(stderr, "\t- e.g. \"XX000000X0000X000\"\n\n");
    return -1;
}

/* retrieve data structures from XML file */
if(!parseXMLFile(xml_filename, &instrument)) {
    fprintf(stderr,
        "Impedance error: Impedance failed to parse XML file.\n");
    return -1;
}

/* set the air properties (speed of sound, density) for each
   segment */
setAirProperties(instrument, temp, temp, 0, humid, X_C02);

/* set fingering from holestring and validate */
if(!setFingering(instrument, holestring)) {
    fprintf(stderr, "Impedance error: \"%s\" ", holestring);
    fprintf(stderr,
        "is an invalid fingering for the given woodwind definition.\n");
    return -1;
}

/* for each frequency in spectrum range... */
for(f = flo; f <= fhi; f += fres) {
    /* calculate impedance */
    Z = impedance(f, instrument, entryratio);
    /* print output */
    printf("%e\t%e\t%e\n", f, Z.Re, Z.Im);
}
return 0;
}

int parseCommandLine(int argc, char** argv, char** holestring,
    double* temp, double* humid, double* flo, double* fhi,
    double* fres, double* entryratio, char** xml_filename) {
    int i;
    double d;
    int sflag = 0, tflag = 0, uflag = 0, lflag = 0, hflag = 0,
        rflag = 0, eflag = 0;
    int numoptions = 7, numinputfiles = 1;
    int minargc = 1 + numinputfiles;
    int maxargc = minargc + 2*numoptions;

    /* Check correct number of parameters */
    if((argc < minargc) || (argc%2 != minargc%2) || (argc > maxargc))
        return 0;

    /* Set default options */
    *holestring = NULL;
    *temp = TEMP;
    *humid = HUMID;
    *flo = FLO;
    *fhi = FHI;
    *fres = FRES;
    *entryratio = ENTRYRATIO;
}

```

```
/* Check and set options */
for(i = 1; i < (argc - numinputfiles); i += 2) {
    if(strcmp(argv[i], "-s") == 0) {
        if(sflag)
            return 0;
        *holestring = argv[i + 1];
        sflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-t") == 0) {
        if(tflag)
            return 0;
        *temp = atof(argv[i + 1]);
        if((*temp < 0.0) || (*temp > 30.0)) {
            fprintf(stderr, "Invalid -t option\n");
            return 0;
        }
        tflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-u") == 0) {
        if(uflag)
            return 0;
        *humid = atof(argv[i + 1]);
        if((*humid < 0.0) || (*humid > 1.0)) {
            fprintf(stderr, "Invalid -u option\n");
            return 0;
        }
        uflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-l") == 0) {
        if(lflag)
            return 0;
        *flo = atof(argv[i + 1]);
        if(*flo <= 0.0) {
            fprintf(stderr, "Invalid -l option\n");
            return 0;
        }
        lflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-h") == 0) {
        if(hflag)
            return 0;
        *fhi = atof(argv[i + 1]);
        if(*fhi <= 0.0) {
            fprintf(stderr, "Invalid -h option\n");
            return 0;
        }
        hflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-r") == 0) {
        if(rflag)
            return 0;
        *fres = atof(argv[i + 1]);
        if(*fres <= 0.0) {
```

```
    fprintf(stderr, "Invalid -r option\n");
    return 0;
}
rflag = 1;
continue;
}
if(strcmp(argv[i], "-e") == 0) {
    if(eflag)
        return 0;
    *entryratio = atof(argv[i + 1]);
    if((*entryratio <= 0.0) || (*entryratio > 1.0)) {
        fprintf(stderr, "Invalid -e option\n");
        return 0;
    }
    eflag = 1;
    continue;
}
/* else invalid option */
fprintf(stderr, "Invalid option: %s\n", argv[i]);
return 0;
}

/* Swap frequency low and high values if inverted */
if((*flo > *fhi) && (*flo != 0.0) && (*fhi != 0.0)) {
    d = *flo;
    *flo = *fhi;
    *fhi = d;
}

/* Set XML filename */
*xml_filename = argv[argc - numinputfiles];
return 1;
}
```

Listing B.11: Map.h

```

/*
Map.h
By Paul Dickens, 2005

Map.c is a Linked List implementation of a Map (associative array).
*/
#ifndef MAP_H_PROTECTOR
#define MAP_H_PROTECTOR

#include "Vector.h"

/* Pair: { double key, pointer to value } */
typedef struct Pair_str {
    double key;
    void* value;
} *Pair;

/* Map: { a Vector of mappings. } */
typedef Vector Map;

Map createMap(void);
/*
Initialises a map with no mappings.
Returns:
A Map if successful, NULL otherwise.
*/
void put(Map m, double key, void* value);
/*
Assigns the given value to the given key. If the map already
contains the key, then its value is reset.
Parameters:
m: the Map to put new mapping in.
key: a double key.
value: a pointer to the value.
*/
void* get(Map m, double key);
/*
Returns the data structure pointer within the given map mapped to
the given key.
Parameters:
m: the Map to retrieve from.
key: the key to get the mapping of.
Returns:
The stored void* pointer within the Map.
NULL if the Map does not contain the given key.
*/
int containsKey(Map m, double key);
/*
Returns true if the given Map contains the given key.
Parameters:

```

```
m: the Map to check the keys of.  
key: the key to check.  
Returns:  
    1 if the Map contains the given key.  
    0 otherwise.  
*/  
  
void clear(Map m);  
/*  
    Removes all mappings from the given Map.  
Parameters:  
    m: the Map to clear.  
*/  
  
int sizeMap(Map m);  
/*  
    Returns the number of mappings in the map.  
Parameters:  
    m: the Map.  
Returns:  
    The size of the map.  
*/  
  
#endif
```

Listing B.12: Map.c

```

/*
Map.c
By Paul Dickens, 2005

Map.c is a Linked List implementation of a Map (associative array).
Refer to Map.h for interface details.
*/

#include <stdlib.h>
#include "Map.h"

Map createMap(void) {
    return createVector();
}

void put(Map m, double key, void* value) {
    int i;
    Pair cur;
    Pair newPair;
    /* search for key */
    for(i = 0; i < sizeMap(m); i++) {
        cur = (Pair) elementAt(m, i);
        if((cur->key) == key) {
            cur->value = value;
            return;
        }
    }
    /* if key not found, make new entry */
    newPair = malloc(sizeof(*newPair));
    newPair->key = key;
    newPair->value = value;
    addElement(m, newPair);
}

void* get(Map m, double key) {
    int i;
    Pair cur;
    /* search for key */
    for(i = 0; i < sizeMap(m); i++) {
        cur = (Pair) elementAt(m, i);
        if((cur->key) == key)
            return cur->value;
    }
    /* if key not found, return NULL */
    return NULL;
}

int containsKey(Map m, double key) {
    int i;
    Pair cur;
    /* search for key */
    for(i = 0; i < sizeMap(m); i++) {
        cur = (Pair) elementAt(m, i);
        if((cur->key) == key)
            return 1;
    }
}

```

```
}

/* if key not found, return false */
return 0;
}

void clear(Map m) {
    while (sizeMap(m) > 0)
        popFront(m);
}

int sizeMap(Map m) {
    return sizeVector(m);
}
```

Listing B.13: Minima.h

```

/*
Minima.h
By Andrew Botros, 2001-2004.
Modified by Paul Dickens, 2006-2007.

Characterises the minima of acoustic impedance spectra.

IMPORTANT: Calculations are optimised for a spectrum resolution of
2Hz.

NOTE: The weighting of the frequency of an impedance extremum fit to
favour the absolute extremum has been changed to an option in the
function 'parabolaExt' with a default of 0 (no weighting). A block
has also been added to the function 'extrema' to handle the case
when the next point to be considered is equal to the previous one
(this fixed a bug in the code).
*/
#endif MINIMA_H_PROTECTOR
#define MINIMA_H_PROTECTOR

#include "Vector.h"
#include "Note.h"

/* minmax: a minimum/maximum flag type */
typedef enum {MINIMUM, MAXIMUM} minmax;

/*
Minimum: {
    musical note struct, frequency, impedance, bandwidth,
    number of harmonics, weighted average harmonic impedance,
    distance to left minimum in Hz,
    difference in impedance of left minimum in dB,
    distance to right minimum in Hz,
    difference in impedance of right minimum in dB,
    distance to left maximum in Hz,
    difference in impedance of left maximum in dB,
    distance to right maximum in Hz,
    difference in impedance of right maximum in dB
}
*/
typedef struct minimum_str {
    Note note;
    double f;
    double Z;
    double B;
    double numharm;
    double meanharmZ;
    double L_min_df;
    double L_min_dZ;
    double R_min_df;
    double R_min_dZ;
    double L_max_df;
}

```

```

    double L_max_dZ;
    double R_max_df;
    double R_max_dZ;
} *Minimum;

/* Extremum: { max/min, frequency, impedance, bandwidth } */
typedef struct extremum_str {
    minmax type;
    double f;
    double Z;
    double B;
} *Extremum;

/* Harmonic:
   { harmonic number, weighted average harmonic impedance } */
typedef struct harmonic_str {
    int n;
    double Z;
} *Harmonic;

Vector minima(Vector allpoints);
/*
   Evaluates the minima in an impedance spectra data file.
   Parameters:
       allpoints: the vector of data points.
   Returns:
       A vector of Minimum structs.
*/
Vector extrema(Vector allpoints);
/*
   Evaluates the extrema in an impedance spectra data file.
   Parameters:
       allpoints: the vector of data points.
   Returns:
       A vector of Extremum structs, or NULL if bad data file.
*/
Extremum parabolaExt(Vector points, minmax type, int weight);
/*
   Calculates the extremum of a given vector of data points. It
   performs a least squares parabolic fit on the data and evaluates the
   frequency, impedance and bandwidth from this fit. Note that the
   frequency of extrema are (optionally) averaged with the absolute
   extrema present in the data set.
   Parameters:
       points: a Vector of Points about the extremum.
       type: a minmax type indicating minimum/maximum.
       weight: if true weights the calculation of the extremum frequency
               in favour of the absolute extremum.
   Returns:
       An Extremum struct containing evaluated minimum.
*/
int progressPoints(Vector points, Vector allpoints, int index);

```

```

/*
Reads the next data point present in the impedance spectra file, and
places this within a window of data points - also shifting the
window to the right by one point and popping the leftmost data
point.
Parameters:
    points: the window of data points that the function updates.
    allpoints: the vector of all data points.
    index: the next point in the allpoints vector to read.
Returns:
    1 if successful, 0 otherwise (such as EOF reached).
*/
int numExtrema(Vector points, minmax type);
/*
Calculates the number of local extrema in the given Vector window.
Parameters:
    points: the window of data points that the function processes.
    type: a minmax type indicating minimum/maximum.
Returns:
    The number of local extrema.
*/
double lowest(Vector points);
/*
Evaluates the frequency of the lowest y value among a set of data
points.
Parameters:
    points: a Vector of Points
Returns:
    A double of the frequency of the lowest y value among points
*/
double highest(Vector points);
/*
Evaluates the frequency of the highest y value among a set of data
points.
Parameters:
    points: a Vector of Points
Returns:
    A double of the frequency of the highest y value among points
*/
void harmonicity(Vector minv);
/*
Sets the harmonicity variables numharm and meanharmZ
for each Minimum struct in a given minima vector.
Parameters:
    minv: the vector of Minimum structs to be updated
*/
Vector harmonics(Vector minv, int pos);
/*
Evaluates the harmonics of a particular Minimum struct in a
given minima vector.
Parameters:
    minv: the vector of Minimum structs for a particular data file
    pos: the index of the required Mimimum struct in the vector

```

```

    (starting at 0)
>Returns:
A vector of Harmonic structs, one for each existent harmonic for
the particular Minimum in question.
*/
double harmAvgZ(Vector harmv);
/*
Calculates the weighted average of the impedance levels for the
first HARMONICS_AVERAGED number of harmonics in a given set. Each
harmonic in the vector contains its integer frequency ratio
(relative to an anonymous minimum), and its impedance level. An
integer ratio of 2 (the second harmonic) has a relative weight of
1/2. An integer ratio of 3 (the third harmonic) has a relative
weight of 1/3, and so on.
>Parameters:
harmv: the vector of Harmonic structs for a particular minimum
>Returns:
The weighted average of their impedance levels as a double
*/
double* harmonicsArray(Vector minv, int pos);
/*
Calculates the frequency ratios of minima following a particular
Minimum.
>Parameters:
minv: the vector of Minimum structs for a particular data file
pos: the index of the required Minimum struct in the vector
(starting at 0)
>Returns:
An array of doubles, one for each minimum following the Minimum
in question. Each double is the frequency of the higher minima
divided by the frequency of the Minimum in question.
*/
double invalidNum(void);
/*
Gives the result of atof("+NAN").
>Returns:
A double representing NaN.
*/
int isValidNum(double d);
/*
Evaluates whether a number is valid. NaN and inf values are invalid.
>Parameters:
d: the double to evaluate
>Returns:
1 if valid, 0 otherwise
*/
double det(double** M);
/*
Calculates the determinant of a 3x3 matrix.
>Parameters:
M: a 2 dimensional double array representing the matrix.
>Returns:
The determinant of M as a double.

```

```
*/  
  
double** A(Vector points);  
double** B(Vector points);  
double** C(Vector points);  
double** D(Vector points);  
/*  
   A, B, C, D return matrices necessary in calculating the least  
   squares parabolic fit to a set of data points. Each function is  
   passed a Vector of Points about the minimum to be fitted.  
  
   Refer to Bevington (1969), "Data Reduction and Error Analysis  
   for the Physical Sciences".  
*/  
  
#endif
```

Listing B.14: Minima.c

```

/*
Minima.c
By Andrew Botros, 2001-2004.
Modified by Paul Dickens, 2006-2007.

Characterises the minima of acoustic impedance spectra.
Refer to Minima.h for interface details.

IMPORTANT: Calculations are optimised for a spectrum resolution of
2Hz.

NOTE: The weighting of the frequency of an impedance extremum fit to
favour the absolute emtremum has been changed to an option in the
function 'parabolaExt' with a default of 0 (no weighting). A block
has also been added to the function 'extrema' to handle the case
when the next point to be considered is equal to the previous one
(this fixed a bug in the code).
*/

```

```

/* necessary define for some gcc math.h functions */
#ifndef _ISOC99_SOURCE
#define _ISOC99_SOURCE
#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "Vector.h"
#include "Point.h"
#include "Note.h"
#include "Minima.h"

/* number of data points approximated around each minimum */
#define NUM_POINTS 11
/* number of points which must be consecutively increasing or
decreasing to define the spectrum as rising or dropping. */
#define TWEAK 5
/* weight fit to absolute extremum? */
#define WEIGHT 0
/* to evaluate harmonics, a window is defined around the frequency f
of the fundamental minimum: f +/- HARMONIC_WINDOW/100. Minima which
lie within integer multiples of this window are deemed to be
harmonics. */
#define HARMONIC_WINDOW 5.0
/* maximum number of harmonics used to calculate average impedance of
harmonics */
#define HARMONICS_AVERAGED 3

Vector minima(Vector allpoints) {
    int i, j;
    Vector extv = extrema(allpoints);
    Extremum e, searche;
```

```

Minimum m;
Vector minv = createVector();

/* for each minimum in extrema list */
for(i = 0; i < sizeVector(extv); i++) {
    e = (Extremum)elementAt(extv, i);
    if(e->type == MINIMUM) {
        /* create Minimum struct with same f, Z, B */
        m = (Minimum)malloc(sizeof(*m));
        m->f = e->f;
        m->Z = e->Z;
        m->B = e->B;
        /* search for left maximum and calculate df, dZ */
        for(j = i - 1; j >= 0; j--) {
            searche = (Extremum)elementAt(extv, j);
            if(searche->type == MAXIMUM) {
                m->L_max_df = e->f - searche->f;
                m->L_max_dZ = searche->Z - e->Z;
                break;
            }
        }
        /* if none found, set to nan */
        if(j < 0) {
            m->L_max_df = invalidNum();
            m->L_max_dZ = invalidNum();
        }
        /* search for left minimum and calculate df, dZ */
        for(j = i - 1; j >= 0; j--) {
            searche = (Extremum)elementAt(extv, j);
            if(searche->type == MINIMUM) {
                m->L_min_df = e->f - searche->f;
                m->L_min_dZ = searche->Z - e->Z;
                break;
            }
        }
        /* if none found, set to nan */
        if(j < 0) {
            m->L_min_df = invalidNum();
            m->L_min_dZ = invalidNum();
        }
        /* search for left maximum and calculate df, dZ */
        for(j = i + 1; j < sizeVector(extv); j++) {
            searche = (Extremum)elementAt(extv, j);
            if(searche->type == MAXIMUM) {
                m->R_max_df = searche->f - e->f;
                m->R_max_dZ = searche->Z - e->Z;
                break;
            }
        }
        /* if none found, set to nan */
        if(j >= sizeVector(extv)) {
            m->R_max_df = invalidNum();
            m->R_max_dZ = invalidNum();
        }
        /* search for left minimum and calculate df, dZ */
        for(j = i + 1; j < sizeVector(extv); j++) {
            searche = (Extremum)elementAt(extv, j);
            if(searche->type == MINIMUM) {

```

```

        m->R_min_df = searche->f - e->f;
        m->R_min_dZ = searche->Z - e->Z;
        break;
    }
}
/* if none found, set to nan */
if(j >= sizeVector(extv)) {
    m->R_min_df = invalidNum();
    m->R_min_dZ = invalidNum();
}
/* add minimum to minimum vector */
addElement(minv, m);
}
}

/* set the harmonic details of all minima in vector */
harmonicity(minv);

return minv;
}

Vector extrema(Vector allpoints) {
    int i, j;
    int seq_inc, seq_dec;
    int descent, ascent;
    Extremum e;
    Point p;
    Vector extv = createVector();
    Vector points = createVector();

    /* load vector of points with the initial portion of data.
       All points are equal to the first point in the data. */
    for(i = 0; i < NUM_POINTS; i++) {
        p = (Point)malloc(sizeof(*p));
        *p = *((Point)elementAt(allpoints, 0));
        addElement(points, p);
    }

    /* initialise flags */
    seq_inc = 0;
    seq_dec = 0;
    descent = 0;
    ascent = 0;

    /* progress points one at a time till end of data set */
    for(i = 0; progressPoints(points, allpoints, i); i++) {

        /* if graph increases... */
        if((Point)elementAt(points, sizeVector(points) - 1))->y
            > ((Point)elementAt(points, sizeVector(points) - 2))->y) {

            /* tally number of points increased ... set flag when
               we have moved up far enough to expect minimum/maximum */
            seq_inc+=2;
            if(seq_inc/2 >= TWEAK)
                ascent = 1;
            if(seq_dec/2 > 0)
                seq_dec-=2;
        }
    }
}

```

```

if(descent && ascent) {
    /* run parabola least squares fit on points, add minimum to
       vector */
    for(j = 0; j < (NUM_POINTS/2 - TWEAK); j++) {
        progressPoints(points, allpoints, i);
        i++;
    }
    e = parabolaExt(points, MINIMUM, WEIGHT);
    if(e != NULL)
        addElement(extv, e);
    /* reset flags and remember we're ascending */
    seq_inc = 0;
    seq_dec = 0;
    descent = 0;
    ascent = 1;
}
}

/* if graph decreases... */
if(((Point)elementAt(points, sizeVector(points) - 1))->y
   < ((Point)elementAt(points, sizeVector(points) - 2))->y) {

    /* tally number of points increased ... set flag when
       we have moved up far enough to expect minimum/maximum */
    seq_dec+=2;
    if(seq_dec/2 >= TWEAK)
        descent = 1;
    if(seq_inc/2 > 0)
        seq_inc-=2;

    if(ascent && descent) {
        /* run parabola least squares fit on points, add minimum to
           vector */
        for(j = 0; j < (NUM_POINTS/2 - TWEAK); j++) {
            progressPoints(points, allpoints, i);
            i++;
        }
        e = parabolaExt(points, MAXIMUM, WEIGHT);
        if(e != NULL)
            addElement(extv, e);
        /* reset flags and remember we're descending*/
        seq_inc = 0;
        seq_dec = 0;
        descent = 1;
        ascent = 0;
    }
}

/* if graph neither increases nor decreases... */
if(((Point)elementAt(points, sizeVector(points) - 1))->y
   == ((Point)elementAt(points, sizeVector(points) - 2))->y) {

    if(descent) {
        seq_inc++;
        seq_dec--;
    }
}

```

```

    }
    if(ascent) {
        seq_dec++;
        seq_inc--;
    }
}

return extv;
}

Extremum parabolaExt(Vector points, minmax type, int weight) {
    Extremum ext = (Extremum)malloc(sizeof(*ext));
    int numext;
    double a, b, c, delta;
    double x0, y0;
    double minf;
    double maxf;

    /* compute least squares parabolic fit from data vector */
    delta = det(D(points));
    a = (1.0/delta)*det(A(points));
    b = (1.0/delta)*det(B(points));
    c = (1.0/delta)*det(C(points));

    if(type == MINIMUM) {
        ext->type = MINIMUM;

        /* determine analytical minimum from parabola fit */
        x0 = -1.0*(b/(2.0*c));
        y0 = a + b*x0 + c*x0*x0;

        if(weight) {
            /* find frequency of absolute minimum in data vector */
            minf = lowest(points);

            numext = numExtrema(points, MINIMUM);

            /* if only one minimum in set, heavily weight to absolute
               minimum */
            if(numext == 1) {
                ext->f = (x0*minf)/(0.95*x0 + 0.05*minf);
            }
            /* otherwise, moderately weight towards absolute minimum */
            else {
                ext->f = (x0*minf)/(0.75*x0 + 0.25*minf);
            }
        }
        /* if weight is false, use fit to determine f */
        else
            ext->f = x0;

        /* construct minimum ...
           impedance is parabola minimum
           bandwidth and Q factor is 3dB span of parabola */
        ext->Z = y0;
        ext->B =
    }
}

```

```

2.0*((sqrt((y0 + 3.0 - a)/c
    + (b/(2.0*c))*(b/(2.0*c))) - (b/(2.0*c))) - x0);
}
else {
    ext->type = MAXIMUM;

    /* determine analytical maximum from parabola fit */
    x0 = -1.0*(b/(2.0*c));
    y0 = a + b*x0 + c*x0*x0;

    if(weight) {
        /* find frequency of absolute maximum in data vector */
        maxf = highest(points);

        numext = numExtrema(points, MAXIMUM);

        /* if only one maximum in set, heavily weight to absolute
           maximum */
        if(numext == 1) {
            ext->f = (x0*maxf)/(0.95*x0 + 0.05*maxf);
        }
        /* otherwise, moderately weight towards absolute maximum */
        else {
            ext->f = (x0*maxf)/(0.75*x0 + 0.25*maxf);
        }
    }
    /* if weight is false, use fit to determine f */
    else
        ext->f = x0;

    /* construct maximum
       impedance is parabola maximum
       bandwidth and Q factor is 3dB span of parabola */
    ext->Z = y0;
    ext->B =
        2.0*((sqrt((y0 - 3.0 - a)/c
            + (b/(2.0*c))*(b/(2.0*c))) - (b/(2.0*c))) - x0);
}

return ext;
}

int progressPoints(Vector points, Vector allpoints, int index) {
    Point p = (Point)malloc(sizeof(*p));
    /* read points if data still in file */
    if(index < sizeVector(allpoints)) {
        *p = *((Point)elementAt(allpoints, index));
        /* remove oldest point and add newly read point */
        popFront(points);
        addElement(points, p);
        return 1;
    }
    /* otherwise end of data reached */
    else
        return 0;
}

int numExtrema(Vector points, minmax type) {

```

```

int i;
int nmin = 0;
int nmax = 0;
int descent = 1;
Point p1, p2;
p1 = (Point)elementAt(points, 0);
/* run through all points given */
for(i = 1; i < sizeVector(points); i++) {
    p2 = (Point)elementAt(points, i);
    /* if next point increases */
    if(p2->y > p1->y && descent) {
        nmin++;
        descent = 0;
    }
    /* if next point decreases, indicate decrease */
    if(p2->y < p1->y && !descent) {
        nmax++;
        descent = 1;
    }
    p1 = p2;
}
if(type == MINIMUM)
    return nmin;
else
    return nmax;
}

double lowest(Vector points) {
    int i;
    Point p;
    double minf, minZ;
    /* minimum is set to first point */
    p = (Point)elementAt(points, 0);
    minf = p->x;
    minZ = p->y;
    /* find lowest point in data vector and return it */
    for(i = 1; i < sizeVector(points); i++) {
        p = (Point)elementAt(points, i);
        if(p->y < minZ) {
            minf = p->x;
            minZ = p->y;
        }
    }
    return minf;
}

double highest(Vector points) {
    int i;
    Point p;
    double maxf, maxZ;
    /* maximum is set to first point */
    p = (Point)elementAt(points, 0);
    maxf = p->x;
    maxZ = p->y;
    /* find highest point in data vector and return it */
    for(i = 1; i < sizeVector(points); i++) {
        p = (Point)elementAt(points, i);
        if(p->y > maxZ) {

```

```

    maxf = p->x;
    maxZ = p->y;
}
}
return maxf;
}

void harmonicity(Vector minv) {
    int i;
    Vector harmv;
    /* for each Minimum in vector... */
    for(i = 0; i < sizeVector(minv); i++) {
        /* find all the harmonic minima of the particular Minimum */
        harmv = harmonics(minv, i);
        /* record the number of harmonics the Minimum has */
        ((Minimum)elementAt(minv, i))->numharm =
        (double)sizeVector(harmv);
        /* calculate and record the weighted average impedance of
         these harmonic minima. */
        ((Minimum)elementAt(minv, i))->meanharmZ = harmAvgZ(harmv);
    }
    return;
}

Vector harmonics(Vector minv, int pos) {
    int i;
    int harmonic;
    double window, left_bound, right_bound;
    double df;
    int curharmonic;
    int found;

    /* get the ratios of minima frequencies following
     the Minimum in question */
    double* ha = harmonicsArray(minv, pos);
    int arraysize = sizeVector(minv) - pos - 1;

    Vector harmv = createVector();
    Harmonic h;

    i = 1;

    /* search for possible integer harmonics until
     last Minimum is reached */
    for(harmonic = 2; ; harmonic++) {
        i--;

        /* calculate a window in which harmonic will be defined.
         The widest window allowable is 0.5. */
        window = (HARMONIC_WINDOW/100.0)*harmonic;
        if(window > 0.5)
            window = 0.5;
        left_bound = harmonic - window;
        right_bound = harmonic + window;
        found = 0;

        /* search for any Minimum which exists in the defined window.
         Compare this to previously found minima in the same

```

```

window, and choose the Minimum closest to the integer
mark as the harmonic. */
while(i < arraysize) {
    if(ha[i] >= right_bound) {
        i++;
        break;
    } else {
        if((ha[i] > left_bound) && (ha[i] < right_bound)) {
            if(found) {
                if(fabs(harmonic - ha[i]) < df) {
                    df = fabs(harmonic - ha[i]);
                    curharmonic = i;
                }
            } else {
                df = fabs(harmonic - ha[i]);
                curharmonic = i;
                found = 1;
            }
        }
        i++;
    }
}

/* if a minimum is found in this window, enter the rounded off
   integer ratio and the impedance level of the harmonic into
   a Harmonic structure and insert this into a vector. */
if(found && (ha[i - 1] >= harmonic)) {
    h = (Harmonic)malloc(sizeof(*h));
    h->n = round(ha[curharmonic]);
    h->Z = ((Minimum)elementAt(minv, pos + curharmonic + 1))->Z;
    addElement(harmv, h);
}

/* if last Minimum reached, break */
if(i == arraysize)
    break;
}

/* return all found harmonics */
return harmv;
}

double harmAvgZ(Vector harmv) {
    int i;
    int numharm;
    double totalZ = 0.0;
    double totalfraction = 0.0;
    Harmonic h;

    /* maximum number of harmonics averaged
       is either HARMONICS_AVERAGED or the number
       of harmonics present */
    if(sizeVector(harmv) < HARMONICS_AVERAGED)
        numharm = sizeVector(harmv);
    else
        numharm = HARMONICS_AVERAGED;

    /* if no harmonics present, return NaN */
}

```

```

if(numharm == 0)
    return invalidNum();

/* for the first few appropriate harmonics,
   compute and return the weighted sum of their impedance
   levels, with the second harmonic (if present) having a
   weighting of 1/2, the third harmonic (if present)
   having a weighting of 1/3, etc. */
for(i = 0; i < numharm; i++) {
    h = (Harmonic)elementAt(harmv, i);
    totalZ = totalZ + (h->Z)/(h->n);
    totalfraction = totalfraction + 1.0/(double)(h->n);
}
return totalZ/totalfraction;
}

double* harmonicsArray(Vector minv, int pos) {
    int i, j = 0;
    int length = sizeVector(minv) - pos - 1;
    double* ha = (double*)malloc(length*sizeof(double));
    double f;
    Minimum m;
    /* get Minimum at required position in vector */
    m = (Minimum)elementAt(minv, pos);
    f = m->f;
    /* for each Minimum following the chosen Minimum,
       calculate frequency ratio of the two minima and
       place result in double array. */
    for(i = pos + 1; i < sizeVector(minv); i++) {
        m = (Minimum)elementAt(minv, i);
        ha[j] = m->f/f;
        j++;
    }
    return ha;
}

double invalidNum(void) {
    return atof("+NAN");
}

int isValidNum(double d) {
    int flag = 1;
    char* plusnantest = (char*)malloc(BUFSIZ);
    char* minusnantest = (char*)malloc(BUFSIZ);
    char* dstring = (char*)malloc(BUFSIZ);

    /* generate all possible floating point errors and
       compare these errors to the given number */
    sprintf(plusnantest, "%f", atof("+NAN"));
    sprintf(minusnantest, "%f", atof("-NAN"));
    sprintf(dstring, "%f", d);
    if(strcmp(dstring, plusnantest) == 0)
        flag = 0;
    if(strcmp(dstring, minusnantest) == 0)
        flag = 0;
    if(d == atof("+INF"))
        flag = 0;
    if(d == atof("-INF"))
        flag = 0;
}

```

```

    flag = 0;
    return flag;
}

/* Refer to Minima.h for documentation on the following */

double det(double** M) {
    return (
        (M[1][1]*(M[2][2]*M[3][3] - M[2][3]*M[3][2])) -
        (M[2][1]*(M[1][2]*M[3][3] - M[1][3]*M[3][2])) +
        (M[3][1]*(M[1][2]*M[2][3] - M[1][3]*M[2][2])));
}

double** A(Vector points) {
    double** M;
    double x, y;
    int i, j;

    M = (double**)malloc(4*sizeof(double*));
    for(i = 0; i <= 3; i++)
        M[i] = (double*)malloc(4*sizeof(double));

    for(i = 1; i <= 3; i++) {
        for(j = 1; j <= 3; j++)
            M[i][j] = 0.0;
    }

    for(i = 0; i < sizeVector(points); i++) {
        x = ((Point)elementAt(points, i))->x;
        y = ((Point)elementAt(points, i))->y;
        M[1][1] += y;
        M[1][2] += x;
        M[1][3] += x*x;
        M[2][1] += x*y;
        M[2][2] += x*x;
        M[2][3] += x*x*x;
        M[3][1] += x*x*y;
        M[3][2] += x*x*x;
        M[3][3] += x*x*x*x;
    }
    return M;
}

double** B(Vector points) {
    double** M;
    double x, y;
    int i, j;

    M = (double**)malloc(4*sizeof(double*));
    for(i = 0; i <= 3; i++)
        M[i] = (double*)malloc(4*sizeof(double));

    for(i = 1; i <= 3; i++) {
        for(j = 1; j <= 3; j++)
            M[i][j] = 0.0;
    }

    for(i = 0; i < sizeVector(points); i++) {

```

```

x = ((Point)elementAt(points, i))->x;
y = ((Point)elementAt(points, i))->y;
M[1][1] += 1.0;
M[1][2] += y;
M[1][3] += x*x;
M[2][1] += x;
M[2][2] += x*y;
M[2][3] += x*x*x;
M[3][1] += x*x;
M[3][2] += x*x*y;
M[3][3] += x*x*x*x;
}
return M;
}

double** C(Vector points) {
    double** M;
    double x, y;
    int i, j;

    M = (double**)malloc(4*sizeof(double*));
    for(i = 0; i <= 3; i++)
        M[i] = (double*)malloc(4*sizeof(double));

    for(i = 1; i <= 3; i++) {
        for(j = 1; j <= 3; j++)
            M[i][j] = 0.0;
    }

    for(i = 0; i < sizeVector(points); i++) {
        x = ((Point)elementAt(points, i))->x;
        y = ((Point)elementAt(points, i))->y;
        M[1][1] += 1.0;
        M[1][2] += x;
        M[1][3] += y;
        M[2][1] += x;
        M[2][2] += x*x;
        M[2][3] += x*y;
        M[3][1] += x*x;
        M[3][2] += x*x*x;
        M[3][3] += x*x*y;
    }
    return M;
}

double** D(Vector points) {
    double** M;
    double x, y;
    int i, j;

    M = (double**)malloc(4*sizeof(double*));
    for(i = 0; i <= 3; i++)
        M[i] = (double*)malloc(4*sizeof(double));

    for(i = 1; i <= 3; i++) {
        for(j = 1; j <= 3; j++)
            M[i][j] = 0.0;
    }
}

```

```
for(i = 0; i < sizeVector(points); i++) {
    x = ((Point)elementAt(points, i))->x;
    y = ((Point)elementAt(points, i))->y;
    M[1][1] += 1.0;
    M[1][2] += x;
    M[1][3] += x*x;
    M[2][1] += x;
    M[2][2] += x*x;
    M[2][3] += x*x*x;
    M[3][1] += x*x;
    M[3][2] += x*x*x;
    M[3][3] += x*x*x*x;
}
return M;
}
```

Listing B.15: ModernFlute.xml

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE woodwind SYSTEM "woodwind.dtd">
<woodwind description="Pearl flute PF-661 open hole">
  <embouchurehole>
    <radiusin>6.4</radiusin>
    <radiusout>5.7</radiusout>
    <length>4.4</length>
    <boreradius>8.533587786259544</boreradius>
  </embouchurehole>
  <upstream flange="-1.0">
    <bore>
      <radius1>8.533587786259544</radius1>
      <radius2>8.4</radius2>
      <length>17.5</length>
    </bore>
  </upstream>
  <downstream flange="0.1">
    <bore>
      <radius1>8.533587786259544</radius1>
      <radius2>9.4</radius2>
      <length>113.5</length>
    </bore>
    <bore>
      <radius1>9.4</radius1>
      <radius2>9.4</radius2>
      <length>37.0</length>
    </bore>
    <bore>
      <radius1>9.9</radius1>
      <radius2>9.9</radius2>
      <length>4.0</length>
    </bore>
    <bore>
      <radius1>9.5</radius1>
      <radius2>9.5</radius2>
      <length>46.2</length>
    </bore>
    <hole name="Tr1">
      <radius>3.7</radius>
      <length>1.9</length>
      <boreradius>9.5</boreradius>
      <key>
        <radius>6.5</radius>
        <holeradius>0.0</holeradius>
        <height>1.6</height>
        <thickness>3.6</thickness>
        <wallThickness>1.0</wallThickness>
        <chimneyHeight>1.5</chimneyHeight>
      </key>
    </hole>
    <bore>
      <radius1>9.5</radius1>
      <radius2>9.5</radius2>
      <length>17.3</length>
    </bore>
    <hole name="Tr2">
      <radius>3.9</radius>

```

```
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
  <radius>6.5</radius>
  <holeradius>0.0</holeradius>
  <height>1.6</height>
  <thickness>3.6</thickness>
  <wallThickness>1.0</wallThickness>
  <chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>16.4</length>
</bore>
<hole name="C#">
  <radius>3.5</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>6.5</radius>
    <holeradius>0.0</holeradius>
    <height>1.9</height>
    <thickness>3.6</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>32.5</length>
</bore>
<hole name="C">
  <radius>6.6</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>9.5</radius>
    <holeradius>0.0</holeradius>
    <height>2.6</height>
    <thickness>3.6</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>19.2</length>
</bore>
<hole name="B">
  <radius>6.6</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>9.5</radius>
```

```
<holeradius>0.0</holeradius>
<height>2.2</height>
<thickness>4.0</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>21.0</length>
</bore>
<hole name="A#">
<radius>6.6</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>3.4</holeradius>
<height>2.5</height>
<thickness>4.2</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>23.2</length>
</bore>
<hole name="A">
<radius>6.6</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>3.4</holeradius>
<height>2.4</height>
<thickness>4.2</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>21.7</length>
</bore>
<hole name="G#">
<radius>6.7</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>0.0</holeradius>
<height>2.4</height>
<thickness>4.0</thickness>
<wallThickness>1.0</wallThickness>
```

```
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>0.5</length>
</bore>
<hole name="G# dup.">
<radius>6.7</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>0.0</holeradius>
<height>2.5</height>
<thickness>4.0</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>24.4</length>
</bore>
<hole name="G">
<radius>7.1</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>0.0</holeradius>
<height>2.1</height>
<thickness>4.0</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>25.7</length>
</bore>
<hole name="F#">
<radius>7.1</radius>
<length>1.9</length>
<boreradius>9.5</boreradius>
<key>
<radius>9.5</radius>
<holeradius>3.4</holeradius>
<height>2.3</height>
<thickness>4.2</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
```

```
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>27.3</length>
</bore>
<hole name="F">
  <radius>7.1</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>9.5</radius>
    <holeradius>3.4</holeradius>
    <height>2.5</height>
    <thickness>4.2</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>28.8</length>
</bore>
<hole name="E">
  <radius>7.1</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>9.5</radius>
    <holeradius>3.4</holeradius>
    <height>2.4</height>
    <thickness>4.2</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>20.7</length>
</bore>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>10.9</length>
</bore>
<hole name="D#">
  <radius>7.7</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>10.3</radius>
    <holeradius>0.0</holeradius>
    <height>2.8</height>
    <thickness>4.0</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
```

```

<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>34.0</length>
</bore>
<hole name="D">
  <radius>7.7</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>10.3</radius>
    <holeradius>0.0</holeradius>
    <height>2.8</height>
    <thickness>4.0</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>
<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>33.0</length>
</bore>
<hole name="C#">
  <radius>7.7</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>10.3</radius>
    <holeradius>0.0</holeradius>
    <height>2.8</height>
    <thickness>4.0</thickness>
    <wallThickness>1.0</wallThickness>
    <chimneyHeight>1.5</chimneyHeight>
  </key>
</hole>


<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>43.1</length>
</bore>



<bore>
  <radius1>9.5</radius1>
  <radius2>9.5</radius2>
  <length>35.0</length>
</bore>
<hole name="C">
  <radius>7.7</radius>
  <length>1.9</length>
  <boreradius>9.5</boreradius>
  <key>
    <radius>10.3</radius>
    <holeradius>0.0</holeradius>

```

```
<height>2.8</height>
<thickness>4.0</thickness>
<wallThickness>1.0</wallThickness>
<chimneyHeight>1.5</chimneyHeight>
</key>
</hole>
<bore>
<radius1>9.5</radius1>
<radius2>9.5</radius2>
<length>41.0</length>
</bore>
<!-- end of code for B foot -->

</downstream>
</woodwind>
```

Listing B.16: Note.h

```

/*
Note.h
By Andrew Botros, 2001-2004
Modified by Paul Dickens, 2007

Note.c is a frequency to musical note converter.

*/
#ifndef NOTE_H_PROTECTOR
#define NOTE_H_PROTECTOR

/* Note: { note struct including name, cents, midi number } */
typedef struct note_str {
    char* name;
    int cents;
    int midi;
} *Note;

Note note(double input, int round);
/*
    Converts the given frequency to a musical note.
    Parameters:
        input: must be a double between 27.5Hz (A0) and 14080Hz (A9)
        round: round note to nearest 5 cents if true
    Returns:
        A note struct which includes:
        - name: A string representing the note, such as "C4"
        - cents: Number of cents as an int, optionally rounded to the
            nearest 5 cents
        - midi: a midi number corresponding to the closest semitone for
            the frequency
        ... OR NULL if frequency out of range.
*/
char* noteString(Note n);
/*
    Returns a string representation of the note.
    Parameters:
        n: The note struct pointer
    Returns:
        A string representation, such as "C4 plus 20 cents" or "A#4 minus
        15 cents"
*/
#endif

```

Listing B.17: Note.c

```

/*
Note.c
By Andrew Botros, 2001-2004
Modified by Paul Dickens, 2007

Note.c is a frequency to musical note converter.
Refer to Note.h for interface details.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Note.h"

#define A4 440.0 /* A4 set at 440Hz reference point */
#define A4_INDEX 57
#define A4_MIDI_INDEX 69

/* array of notes */
char* notes[120] = {
    "C0", "C#0", "D0", "D#0", "E0", "F0", "F#0", "G0", "G#0", "A0", "A#0", "B0",
    "C1", "C#1", "D1", "D#1", "E1", "F1", "F#1", "G1", "G#1", "A1", "A#1", "B1",
    "C2", "C#2", "D2", "D#2", "E2", "F2", "F#2", "G2", "G#2", "A2", "A#2", "B2",
    "C3", "C#3", "D3", "D#3", "E3", "F3", "F#3", "G3", "G#3", "A3", "A#3", "B3",
    "C4", "C#4", "D4", "D#4", "E4", "F4", "F#4", "G4", "G#4", "A4", "A#4", "B4",
    "C5", "C#5", "D5", "D#5", "E5", "F5", "F#5", "G5", "G#5", "A5", "A#5", "B5",
    "C6", "C#6", "D6", "D#6", "E6", "F6", "F#6", "G6", "G#6", "A6", "A#6", "B6",
    "C7", "C#7", "D7", "D#7", "E7", "F7", "F#7", "G7", "G#7", "A7", "A#7", "B7",
    "C8", "C#8", "D8", "D#8", "E8", "F8", "F#8", "G8", "G#8", "A8", "A#8", "B8",
    "C9", "C#9", "D9", "D#9", "E9", "F9", "F#9", "G9", "G#9", "A9", "A#9", "B9" };

#define MINUS 0
#define PLUS 1

Note note(double input, int round) {
    double frequency;
    int r_index = 0;
    int cent_index = 0;
    int side;
    Note n = (Note)malloc(sizeof(Note*));

    /* a semitone higher than a given frequency is  $2^{(1/12)}$  times the
     * frequency
     * a cent higher than a given frequency is  $2^{(1/1200)}$  times the
     * frequency */
    double r = pow(2, 1.0/12.0);
    double cent = pow(2, 1.0/1200.0);

    /* input frequency must be between A0 and A9 */
    if((input < 26.73) || (input > 14496.0)) {
        return NULL;
    }

    /* set A4 (440Hz) as reference point */

```

```

frequency = A4;

/* search for input ratio against A4 to the nearest cent
   in range -49 to +50 cents around closest note */
if(input >= frequency) {
    while(input >= r*frequency) {
        frequency = r*frequency;
        r_index++;
    }
}
else {
    while(input < frequency) {
        frequency = frequency/r;
        r_index--;
    }
}
while(input >= cent*frequency) {
    frequency = cent*frequency;
    cent_index++;
}
if((cent*frequency - input) < (input - frequency))
    cent_index++;
if(cent_index > 50) {
    r_index++;
    cent_index = 100 - cent_index;
    if(cent_index != 0)
        side = MINUS;
    else
        side = PLUS;
}
else
    side = PLUS;

if(round) {
    /* round cents to nearest 5 cents */
    switch(cent_index%5) {
        case 1:
            cent_index = cent_index - 1;
            break;
        case 2:
            cent_index = cent_index - 2;
            break;
        case 3:
            cent_index = cent_index + 2;
            break;
        case 4:
            cent_index = cent_index + 1;
            break;
        default:
            break;
    }
    if(cent_index == 0)
        side = PLUS;
}

/* fill string buffer with note information and return */
n->name = (char*)malloc(BUFSIZ*sizeof(char));
sprintf(n->name, "%s", notes[A4_INDEX + r_index]);
}

```

```
if(side == PLUS)
    n->cents = cent_index;
else
    n->cents = (-1)*cent_index;
n->midi = A4_MIDI_INDEX + r_index;
return n;
}

char* noteString(Note n) {
    char* note_string = (char*)malloc(BUFSIZ*sizeof(char));
    if(n->cents >= 0)
        sprintf(note_string, "%s plus %d cents", n->name, abs(n->cents));
    else
        sprintf(note_string, "%s minus %d cents", n->name, abs(n->cents));
    return note_string;
}
```

Listing B.18: ParseImpedance.h

```
/*
ParseImpedance.h
By Andrew Botros, 2001-2004
Modified by Paul Dickens Apr 2005, 2006

ParseImpedance.c reads an acoustic impedance spectra file and
returns a Vector of Vectors of Points.
*/
#ifndef PARSEIMPEDANCE_H_PROTECTOR
#define PARSEIMPEDANCE_H_PROTECTOR

#include "Vector.h"

int parseImpedanceFile(Vector filev, char* filename);
/*
Reads each line of a given acoustic impedance spectra file
and constructs a Vector of Vectors of Points
representing the file.
Parameters:
    filev: the return Vector
    filename: the name of the file to read
Returns:
    1 if the operation was successful
    0 otherwise
*/
#endif
```

Listing B.19: ParseImpedance.c

```

/*
ParseImpedance.c
By Andrew Botros, 2001-2004
Modified by Paul Dickens, 2005, 2006

ParseImpedance.c reads an acoustic impedance spectra file and
returns a Vector of Vectors of Points. Refer to ParseImpedance.h for
interface details.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Vector.h"
#include "Point.h"
#include "ParseImpedance.h"

int parseImpedanceFile(Vector filev, char* filename) {
    FILE* fp;
    char* line = (char*)malloc(BUFSIZ*sizeof(char));
    char* delimiters = " \t";
    char* token;
    int* midi;
    double x;
    Point p;
    Vector seriesVector;
    int series;

    /* open data file and indicate any error */
    if((fp = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "Cannot open file %s\n", filename);
        return 0;
    }

    /* parse midi line and add midi numbers to Vector */
    fgets(line, BUFSIZ, fp);
    token = strtok(line, delimiters);
    while((token != NULL) && (token != "\n")) {
        seriesVector = createVector();
        midi = (int*)malloc(sizeof(char));
        *midi = atoi(token);
        addElement(seriesVector, midi);
        addElement(filev, seriesVector);
        token = strtok(NULL, delimiters);
    }

    /* parse each line and add points to Vector */
    while(fgets(line, BUFSIZ, fp) != NULL) {
        /* Parse x value */
        token = strtok(line, delimiters);
        if(token != NULL)
            x = atof(token);
        else {
            fprintf(stderr, "File %s is invalid\n", filename);
            return 0;
        }
    }
}

```

```
    }
    series = 0;
    while(((token = strtok(NULL, delimiters)) != NULL)
        && (token != "\n")) {
        p = (Point)malloc(sizeof(*p));
        p->x = x;
        p->y = atof(token);
        seriesVector = (Vector)elementAt(filev, series);
        addElement(seriesVector, p);
        series++;
    }
}
/* close data file and return Vector of Vectors of Points */
fclose(fp);
return 1;
}
```

Listing B.20: ParseXML.h

```

/*
ParseXML.c
By Paul Dickens, 2006
Based on ParseXML.c by Andrew Botros, 2003

ParseXML.c reads a woodwind definition file for Impedance.c
to use in its calculation.

*/
#endif PARSEXML_H_PROTECTOR
#define PARSEXML_H_PROTECTOR

#include <libxml/parser.h>
#include "Vector.h"
#include "Woodwind.h"

/* Elements and attributes of the woodwind schema */
#define WOODWIND "woodwind"
#define DESCRIPTION "description"
#define EMBOUCHUREHOLE "embouchurehole"
#define UPSTREAM "upstream"
#define DOWNSTREAM "downstream"
#define BORE "bore"
#define HOLE "hole"
#define RADIUS "radius"
#define RADIUSIN "radiusin"
#define RADIUSOUT "radiusout"
#define RADIUS1 "radius1"
#define RADIUS2 "radius2"
#define BORERADIUS "boreradius"
#define LENGTH "length"
#define FLANGE "flange"
#define KEY "key"
#define HOLERADIUS "holeradius"
#define HEIGHT "height"
#define THICKNESS "thickness"
#define WALLTHICKNESS "wallthickness"
#define CHIMNEYHEIGHT "chimneyheight"

int parseXMLFile(char* xml_filename, Woodwind* w);
/*
Creates a Woodwind struct after parsing the given XML definition
file.
Parameters:
    xml_filename: filename of the XML file.
    w: to be initialised with data given in XML file.
Returns
    1 if the parse was successful.
    0 otherwise.
*/
int parseAndValidateFile(char* xml_filename, xmlDocPtr* doc);
/*

```

```

Opens XML file and validates it against the (internally defined)
DTD.
Parameters:
    xml_filename: filename of the XML file.
    doc: pointer to xmlDocPtr initialised to xml_filename by libxml.
Returns:
    1 if file is opened correctly and is valid.
    0 otherwise.
*/

int parseWoodwind(xmlDocPtr doc, Woodwind* w);
/*
Creates a Woodwind struct from data in the xml document.
Parameters:
    doc: the parsed xmlDocPtr of the XML document.
    w: to be initialised with data in doc.
Returns:
    1 if all parameters are valid.
    0 otherwise.
*/

int parseEmbouchureHole(xmlDocPtr doc, xmlNodePtr node,
    EmbouchureHole* h);
/*
Creates an EmbouchureHole struct from data in the xml document.
Parameters:
    doc: the parsed xmlDocPtr of the XML document.
    node: the current node.
    h: to be initialised with data in doc.
Returns:
    1 if all parameters are valid.
    0 otherwise.
*/

int parseHole(xmlDocPtr doc, xmlNodePtr node, Hole* h);
/*
Creates a Hole struct from data in the xml document.
Parameters:
    doc: the parsed xmlDocPtr of the XML document.
    node: the current node.
    h: to be initialised with data in doc.
Returns:
    1 if all parameters are valid.
    0 otherwise.
*/

int parseKey(xmlDocPtr doc, xmlNodePtr node, Key* k);
/*
Creates a Key struct from data in the xml document.
Parameters:
    doc: the parsed xmlDocPtr of the XML document.
    node: the current node.
    k: to be initialised with data in doc.
Returns:
    1 if all parameters are valid.
    0 otherwise.
*/

```

```

int parseUpstream(xmlDocPtr doc, xmlNodePtr node, Vector* bore,
  double* flange);
/*
  Creates a TerminalSection of BoreSegments from data in the xml
  document.
Parameters:
  doc: the parsed xmlDocPtr of the XML document.
  node: the current node.
  bore: to be initialised with data in doc.
  flange: the impedance condition at the end of the pipe.
Returns:
  1 if all parameters are valid.
  0 otherwise.
*/
int parseDownstream(xmlDocPtr doc, xmlNodePtr node, Vector* bore,
  Vector* cells, double* flange);
/*
  Creates a Vector of BoreSegments and a Vector of UnitCells
  from data in the xml document.
Parameters:
  doc: the parsed xmlDocPtr of the XML document.
  node: the current node.
  bore: to be initialised with data in doc.
  cells: to be initialised with data in doc.
  flange: the impedance condition at the end of the pipe.
Returns:
  1 if all parameters are valid.
  0 otherwise.
*/
int parseBore(xmlDocPtr doc, xmlNodePtr node, BoreSegment* s);
/*
  Creates a BoreSegment struct from data in the xml document.
Parameters:
  doc: the parsed xmlDocPtr of the XML document.
  node: the current node.
  s: to be initialised with data in doc.
Returns:
  1 if all parameters are valid.
  0 otherwise.
*/
xmlNodePtr getAndAssertDocRoot(xmlDocPtr doc);
/*
  Retrieves the root node of the XML document,
  asserting that it is a <woodwind> element.
Parameters:
  doc: the parsed xmlDocPtr of the XML document.
Returns:
  An xmlNodePtr, pointing at the root node.
  NULL if <woodwind> root element is not found.
*/
int isXMLElement(xmlNodePtr node, const char* expectedname);
/*
  Determines whether the supplied element node is the same type
  as the expected element name.

```

```

Parameters:
  node: the supplied xmlNodePtr.
  expected_name: the expected element name.
Returns:
  1 if names are equivalent.
  0 otherwise.
*/
int assertXMLElement(xmlNodePtr node, const char* expectedname);
/*
Asserts that the supplied element node is the same type
as the expected element name.
Parameters:
  node: the supplied xmlNodePtr.
  expectedname: the expected element name.
Returns:
  1 if names are equivalent.
  0 otherwise (also prints error to stderr).
*/
char* getXMLAttribute(xmlNodePtr node, const char* attributename);
/*
Retrieves the attribute value of a given attribute name within
a given node.
Parameters:
  node: the supplied xmlNodePtr.
  attributename: the supplied attribute name.
Returns:
  The attribute value.
  NULL if attribute does not exist.
*/
double getXMLDoubleAttribute(xmlNodePtr node,
  const char* attributename);
/*
Retrieves the attribute value of a given attribute name within
a given node (as a double).
Parameters:
  node: the supplied xmlNodePtr.
  attributename: the supplied attribute name.
Returns:
  The attribute value.
*/
char* getXMLData(xmlDocPtr doc, xmlNodePtr node);
/*
Retrieves the element data within a given node.
Parameters:
  doc: the supplied xmlDocPtr.
  node: the supplied xmlNodePtr.
Returns:
  The element's value.
*/
int getXMLIntegerData(xmlDocPtr doc, xmlNodePtr node);
/*
Retrieves the integer element data within a given node.
Parameters:

```

```
doc: the supplied xmlDocPtr.  
node: the supplied xmlNodePtr.  
Returns:  
The element's integer value.  
*/  
  
double getXMLDoubleData(xmlDocPtr doc, xmlNodePtr node);  
/*  
Retrieves the floating point element data within a given node.  
Parameters:  
doc: the supplied xmlDocPtr.  
node: the supplied xmlNodePtr.  
Returns:  
The element's double value.  
*/  
  
double getAndScaleXMLDimensionData(xmlDocPtr doc, xmlNodePtr node);  
/*  
Retrieves the dimension (mm) element data within a given node  
and rescales to SI units (m).  
Parameters:  
doc: the supplied xmlDocPtr.  
node: the supplied xmlNodePtr.  
Returns:  
The element's double value.  
*/  
  
#endif
```

Listing B.21: ParseXML.c

```

/*
ParseXML.c
By Paul Dickens, 2006
Based on ParseXML.c by Andrew Botros, 2003

ParseXML.c reads a woodwind definition file for Impedance.c
to use in its calculation.
Refer to ParseXML.h for interface details.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
#include "Vector.h"
#include "Woodwind.h"
#include "ParseXML.h"

int parseXMLFile(char* xml_filename, Woodwind* w) {
    xmlDocPtr doc; /* the resulting document tree */

    /* Read and validate XML file */
    if(!parseAndValidateFile(xml_filename, &doc))
        return 0;

    /* Build instrument */
    if(!parseWoodwind(doc, w))
        return 0;

    /* Free XML data structure */
    xmlFreeDoc(doc);

    return 1;
}

int parseAndValidateFile(char* xml_filename, xmlDocPtr* doc) {
    xmlParserCtxtPtr ctxt; /* the parser context */

    /* create a parser context */
    ctxt = xmlNewParserCtxt();
    if (ctxt == NULL) {
        fprintf(stderr, "XML Error: Failed to allocate parser context\n");
        return 0;
    }

    /* parse the file, activating the DTD validation option */
    *doc = xmlCtxtReadFile(ctxt, xml_filename, NULL,
    XML_PARSE_DTDVALID);
    /* check if parsing succeeded */
    if (*doc == NULL) {
        fprintf(stderr, "XML Error: Failed to parse %s\n", xml_filename);
        return 0;
    }
}

```

```

/* check if validation succeeded */
if (ctxt->valid == 0) {
    fprintf(stderr, "XML Error: Failed to validate %s\n",
    xml_filename);
    return 0;
}
xmlFreeParserCtxt(ctxt);
return 1;
}

int parseWoodwind(xmlDocPtr doc, Woodwind* w) {
    xmlNodePtr curnode;
    EmbouchureHole embouchureHole = NULL;
    Vector upstreamBore, downstreamBore, cells;
    double upstreamFlange, flange;
    Head head;

    /* Retrieve and validate root node */
    if((curnode = getAndAssertDocRoot(doc)) == NULL)
        return 0;

    curnode = curnode->children;
    while(curnode != NULL) {
        if(isXMLElement(curnode, EMBOUCHUREHOLE))
            if(!parseEmbouchureHole(doc, curnode, &embouchureHole))
                return 0;
        if(isXMLElement(curnode, UPSTREAM))
            if(!parseUpstream(doc, curnode, &upstreamBore, &upstreamFlange))
                return 0;
        if(isXMLElement(curnode, DOWNSTREAM))
            if(!parseDownstream(doc, curnode, &downstreamBore, &cells, &
                flange)) return 0;
        curnode = curnode->next;
    }
    head = createHead(embouchureHole, upstreamBore, upstreamFlange,
    downstreamBore);
    *w = createWoodwind(head, cells, flange);
    return 1;
}

int parseEmbouchureHole(xmlDocPtr doc, xmlNodePtr node, EmbouchureHole
* h) {
    double radiusin, radiusout, length, boreRadius;
    node = node->children;
    while(node != NULL) {
        /* Fill Hole data */
        if(isXMLElement(node, RADIUSIN))
            radiusin = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, RADIUSOUT))
            radiusout = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, LENGTH))
            length = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, BORERADIUS))
            boreRadius = getAndScaleXMLDimensionData(doc, node);
        node = node->next;
    }
    /* Ensure dimensions as expected */
    if((radiusin <= 0.0) ||

```

```

(radiusout <= 0.0) ||
(boreRadius <= 0.0) ||
(length <= 0.0)) {
fprintf(stderr,
    "XML error: Embouchure hole has invalid dimensions.\n");
return 0;
}
*h = createEmbouchureHole(radiusin, radiusout, length, boreRadius);
return 1;
}

int parseHole(xmlDocPtr doc, xmlNodePtr node, Hole* h) {
    double radius, length, boreRadius;
    Key key = NULL;
    node = node->children;
    while(node != NULL) {
        /* Fill Hole data */
        if(isXMLElement(node, RADIUS))
            radius = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, LENGTH))
            length = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, BORERADIUS))
            boreRadius = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, KEY))
            if(!parseKey(doc, node, &key)) return 0;
        node = node->next;
    }
    /* Ensure dimensions as expected */
    if((radius <= 0.0) ||
       (boreRadius <= 0.0) ||
       (length <= 0.0)) {
        fprintf(stderr, "XML error: Hole has invalid dimensions.\n");
        return 0;
    }
    *h = createHole(radius, length, boreRadius, key);
    return 1;
}

int parseKey(xmlDocPtr doc, xmlNodePtr node, Key* k) {
    double radius, holeRadius, height, thickness, wallThickness,
    chimneyHeight;
    node = node->children;
    while(node != NULL) {
        /* Fill Key data */
        if(isXMLElement(node, RADIUS))
            radius = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, HOLERADIUS))
            holeRadius = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, HEIGHT))
            height = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, THICKNESS))
            thickness = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, WALLTHICKNESS))
            wallThickness = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, CHIMNEYHEIGHT))
            chimneyHeight = getAndScaleXMLDimensionData(doc, node);
        node = node->next;
    }
}

```

```

/* Ensure dimensions as expected */
if((radius <= 0.0) ||
   (holeRadius < 0.0) ||
   (height <= 0.0) ||
   (thickness <= 0.0) ||
   (wallThickness < 0.0) ||
   (chimneyHeight < 0.0)) {
    fprintf(stderr, "XML error: Key has invalid dimensions.\n");
    return 0;
}
*k = createKey(radius, holeRadius, height, thickness, wallThickness,
chimneyHeight);
return 1;
}

int parseUpstream(xmlDocPtr doc, xmlNodePtr node, Vector* bore,
double* flange) {
BoreSegment s;
*bore = createVector();
*flange = getXMLDoubleAttribute(node, FLANGE);
/* Get current node's children */
node = node->children;
/* Add to bore all segments until the end */
while(node != NULL) {
    if(isXMLElement(node, BORE)) {
        if(!parseBore(doc, node, &s)) return 0;
        addElement(*bore, s);
    }
    node = node->next;
}
/* Ensure flange as expected */
if((*flange < 0.0) &&
   (*flange != -1.0)) {
    fprintf(stderr, "XML error: Flange is invalid.\n");
    return 0;
}
return 1;
}

int parseDownstream(xmlDocPtr doc, xmlNodePtr node, Vector* bore,
Vector* cells, double* flange) {
UnitCell curCell;
BoreSegment s;
Hole h;
Vector curBore;

*bore = createVector();
*cells = createVector();
*flange = getXMLDoubleAttribute(node, FLANGE);
curBore = *bore;
/* Get current node's children */
node = node->children;
while(node != NULL) {
    if(isXMLElement(node, BORE)) {
        if(!parseBore(doc, node, &s)) return 0;
        addElement(curBore, s);
    }
    if(isXMLElement(node, HOLE)) {

```

```

    if(!parseHole(doc, node, &h)) return 0;
    curBore = createVector();
    curCell = createUnitCell(h, curBore);
    addElement(*cells, curCell);
}
node = node->next;
}
/* Ensure flange as expected */
if((*flange < 0.0) &&
   (*flange != -1.0)) {
    fprintf(stderr, "XML error: Flange is invalid.\n");
    return 0;
}
return 1;
}

int parseBore(xmlDocPtr doc, xmlNodePtr node, BoreSegment* s) {
    double radius1, radius2, length;
    /* Search node's children for data elements */
    node = node->children;
    while(node != NULL) {
        if(isXMLElement(node, RADIUS1))
            radius1 = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, RADIUS2))
            radius2 = getAndScaleXMLDimensionData(doc, node);
        if(isXMLElement(node, LENGTH))
            length = getAndScaleXMLDimensionData(doc, node);
        node = node->next;
    }
    /* Ensure dimensions as expected */
    if((radius1 <= 0.0) ||
       (radius2 <= 0.0) ||
       (length <= 0.0)) {
        fprintf(stderr, "XML error: Bore has invalid dimensions.\n");
        return 0;
    }
    *s = createBoreSegment(radius1, radius2, length);
    return 1;
}

xmlNodePtr getAndAssertDocRoot(xmlDocPtr doc) {
    xmlNodePtr rootnode;
    /* Parse root node and ensure XML file is non-empty */
    if((rootnode = xmlDocGetRootElement(doc)) == NULL) {
        fprintf(stderr, "XML error: Empty XML document.\n");
        return NULL;
    }
    /* Ensure root node as expected */
    if(!assertXMLElement(rootnode, WOODWIND)) {
        fprintf(stderr, "XML error: Document root is not <%s>.\n",
               WOODWIND);
        return NULL;
    }
    return rootnode;
}

int isXMLElement(xmlNodePtr node, const char* expectedname) {
    if(xmlStrcmp(node->name, (const xmlChar*)expectedname) == 0)

```

```
    return 1;
else
    return 0;
}

int assertXMLElement(xmlNodePtr node, const char* expectedname) {
    if(isXMLElement(node, expectedname))
        return 1;
    else {
        fprintf(stderr, "XML error: Expected \"%<%s>\", got \"%<%s>\".\n",
                expectedname, (const char*)(node->name));
        return 0;
    }
}

char* getXMLAttribute(xmlNodePtr node, const char* attributename) {
    return (char*)xmlGetProp(node, (const xmlChar*)attributename);
}

double getXMLDoubleAttribute(xmlNodePtr node, const char*
attributename) {
    char* cdata = getXMLAttribute(node, attributename);
    return atof(cdata);
}

char* getXMLData(xmlDocPtr doc, xmlNodePtr node) {
    return (char*)xmlNodeListGetString(doc, node->children, 1);
}

int getXMLIntegerData(xmlDocPtr doc, xmlNodePtr node) {
    char* pcdata = getXMLData(doc, node);
    return atoi(pcdata);
}

double getXMLDoubleData(xmlDocPtr doc, xmlNodePtr node) {
    char* pcdata = getXMLData(doc, node);
    return atof(pcdata);
}

double getAndScaleXMLDimensionData(xmlDocPtr doc, xmlNodePtr node) {
    return 1e-3 * getXMLDoubleData(doc, node);
}
```

Listing B.22: PlayedImpedance.c

```

/*
PlayedImpedance.c
By Paul Dickens, 2006

Physical model of the acoustic impedance of a played flute.
*/



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "Vector.h"
#include "ParseXML.h"


int parseCommandLine(int argc, char** argv, double* flo, double* fhi,
    double* fres, char** input_filename, char** xml_filename);

int parseInputFile(Vector midiv, Vector holestringv,
    char* input_filename);

/* Default spectrum range and resolution */
#define FLO 200.0
#define FHI 4000.0
#define FRE 2.0

int main(int argc, char** argv) {
    double f, flo, fhi, fres;
    char* input_filename;
    char* xml_filename;
    Vector midiv = createVector();
    Vector holestringv = createVector();
    Woodwind instrument;
    int i;
    int midi;
    char* holestring;
    double z_dB;

    /* check correct usage */
    if(!parseCommandLine(argc, argv, &flo, &fhi, &fres, &input_filename,
        &xml_filename)) {
        fprintf(stderr,
            "Usage: PlayedImpedance [OPTIONS] <input file> <XML file>\n\n");
        fprintf(stderr, " Options:\n");
        fprintf(stderr, "\t-l <flo> (default 200.0)\n");
        fprintf(stderr, "\t-h <fhi> (default 4000.0)\n");
        fprintf(stderr, "\t-r <fres> (default 2.0)\n\n");
        fprintf(stderr, " <input file>:\n");
        fprintf(stderr,
            "\t- Must be a tab-delimited list of midi numbers and\n");
        fprintf(stderr, "\t holestrings, one set per line.\n\n");
        return -1;
    }

    /* parse input file */
    if(!parseInputFile(midiv, holestringv, input_filename)) {

```

```

fprintf(stderr, "PlayedImpedance error: ");
fprintf(stderr, "PlayedImpedance failed to parse input file.\n");
return -1;
}

/* retrieve data structures from XML file */
if(!parseXMLFile(xml_filename, &instrument)) {
    fprintf(stderr, "PlayedImpedance error: ");
    fprintf(stderr, "PlayedImpedance failed to parse XML file.\n");
    return -1;
}

discretiseWoodwind(instrument, WW_MAX_LENGTH);
setAirProperties(instrument, WW_T_0, WW_T_AMB, WW_T_GRAD, WW_HUMID,
WW_X_CO2);

/* print the midi numbers as column labels */
for(i = 0; i < sizeVector(midiv); i++) {
    /* set midi from vector */
    midi = atoi((char*)elementAt(midiv, i));
    printf("\t%d", midi);
}
printf("\n");

/* for each frequency in spectrum range... */
for(f = flo; f <= fhi; f += fres) {
    printf("%.2f", f);
    /* for each fingering... */
    for(i = 0; i < sizeVector(midiv); i++) {
        /* print tab delimiter */
        printf("\t");
        /* set midi and holestring from vectors */
        midi = atoi((char*)elementAt(midiv, i));
        holestring = (char*)elementAt(holestringv, i);
        /* set and validate fingering */
        if(!setFingering(instrument, holestring)) {
            fprintf(stderr, "PlayedImpedance error: \"%s\" ",
                (char*)elementAt(holestringv, i));
            fprintf(stderr,
                "is an invalid fingering for the given woodwind ");
            fprintf(stderr, "definition.\n");
            return -1;
        }
        /* calculate and output impedance */
        z_dB = 20.0*log10(modz(playedImpedance(f, instrument, midi)));
        printf("%.3f", z_dB);
    }
    /* print new line */
    printf("\n");
}
return 0;
}

int parseCommandLine(int argc, char** argv, double* flo, double* fhi,
    double* fres, char** input_filename, char** xml_filename) {
    int i;
    double d;
    int lflag = 0, hflag = 0, rflag = 0;

```

```

int numoptions = 3, numinputfiles = 2;
int minargc = 1 + numinputfiles;
int maxargc = minargc + 2*numoptions;

/* Check correct number of parameters */
if((argc < minargc) || (argc%2 != minargc%2) || (argc > maxargc))
    return 0;

/* Set default options */
*flo = FL0;
*fhi = FHI;
*fres = FRES;

/* Check and set options */
for(i = 1; i < (argc - numinputfiles); i += 2) {
    if(strcmp(argv[i], "-l") == 0) {
        if(lflag)
            return 0;
        *flo = atof(argv[i + 1]);
        if(*flo <= 0.0) {
            fprintf(stderr, "Invalid -l option\n");
            return 0;
        }
        lflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-h") == 0) {
        if(hflag)
            return 0;
        *fhi = atof(argv[i + 1]);
        if(*fhi <= 0.0) {
            fprintf(stderr, "Invalid -h option\n");
            return 0;
        }
        hflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-r") == 0) {
        if(rflag)
            return 0;
        *fres = atof(argv[i + 1]);
        if(*fres <= 0.0) {
            fprintf(stderr, "Invalid -r option\n");
            return 0;
        }
        rflag = 1;
        continue;
    }
    /* else invalid option */
    fprintf(stderr, "Invalid option: %s\n", argv[i]);
    return 0;
}

/* Swap frequency low and high values if inverted */
if((*flo > *fhi) && (*flo != 0.0) && (*fhi != 0.0)) {
    d = *flo;
    *flo = *fhi;
    *fhi = d;
}

```

```
}

/* Set input filename and XML filename */
/* open input file */
if((fp = fopen(input_filename, "r")) == NULL)
    return 0;
/* add each line (without newline) to hole string vector */
while(1) {
    line = (char*)malloc(BUFSIZ*sizeof(char));
    if(fgets(line, BUFSIZ, fp) == NULL)
        break;
    token = strtok(line, delimiters);
    addElement(midiv, token);
    token = strtok(NULL, delimiters);
    addElement(holestringv, token);
}
return 1;
}
```

Listing B.23: Point.h

```
/*
Point.h
By Andrew Botros, 2001-2004

A data point ADT.
*/

#ifndef POINT_H_PROTECTOR
#define POINT_H_PROTECTOR

/* Point: { x, y } */
typedef struct point_str {
    double x;
    double y;
} *Point;

void printPoint(Point p);
/*
Prints the contents of a Point struct to stdout.
Parameters:
    p: the Point struct.
*/

void printImpedancePoint(Point p);
/*
Prints the contents of a Point struct to stdout in the Impedance
format.
Parameters:
    p: the Point struct.
*/

Point parseImpedancePoint(char* line, int series);
/*
Parses a line of text into its impedance point data.
Parameters:
    line: the line of data.
Returns:
    A Point struct representing the data.
    NULL if line does not contain two numbers.
*/

#endif
```

Listing B.24: Point.c

```
/*
Point.h
By Andrew Botros, 2001-2004

A data point ADT.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Point.h"

void printPoint(Point p) {
    printf("(%.2f,%.2f)", p->x, p->y);
    return;
}

void printImpedancePoint(Point p) {
    printf("%.2f\t%.2f\n", p->x, p->y);
    return;
}

Point parseImpedancePoint(char* line, int series) {
    Point p = (Point)malloc(sizeof(*p));
    char* delimiters = "\t";
    char* token;
    int n;
    /* Parse x value */
    token = strtok(line, delimiters);
    if(token != NULL)
        p->x = atof(token);
    else
        return NULL;
    /* Parse y value */
    for (n = 0; n < series; n++)
        token = strtok(NULL, delimiters);
    if(token != NULL && token != "\n") {
        p->y = atof(token);
    }
    else
        return NULL;
    return p;
}
```

Listing B.25: TransferMatrix.h

```

/*
TransferMatrix.h
By Paul Dickens, 2005

Complex transfer matrix library.
*/

#ifndef TRANSFERMATRIX_H_PROTECTOR
#define TRANSFERMATRIX_H_PROTECTOR

#include "Complex.h"

/* TransferMatrix: { A, B, C, D } */
typedef struct transferMatrix_str {
    complex A;
    complex B;
    complex C;
    complex D;
} *TransferMatrix;

TransferMatrix createTransferMatrix(complex A, complex B, complex C,
                                    complex D);
/*
Creates a new TransferMatrix given the complex elements
Returns:
The matrix (A, B, C, D).
*/
TransferMatrix identitym();
/*
Returns:
The identity matrix (1, 0, 0, 1).
*/
void lmultm(TransferMatrix m, TransferMatrix mult);
/*
Left-multiplies matrix m with matrix mult.
*/
void rmultm(TransferMatrix m, TransferMatrix mult);
/*
Right-multiplies matrix m with matrix mult.
*/
void invertm(TransferMatrix m);
/*
Inverts matrix m.
*/
complex calcZin(TransferMatrix m, complex Zload);
/*
Calculates the input impedance given TransferMatrix m and load
Zload.
*/

```

```
#endif
```

Listing B.26: TransferMatrix.c

```

/*
TransferMatrix.c
By Paul Dickens, 2005

Complex transfer matrix library.
Refer to TransferMatrix.h for interface details.
*/

#include <stdlib.h>
#include "TransferMatrix.h"

TransferMatrix createTransferMatrix(complex A, complex B, complex C,
complex D) {
    TransferMatrix m = malloc(sizeof(*m));
    m->A = A;
    m->B = B;
    m->C = C;
    m->D = D;
    return m;
}

TransferMatrix identitym() {
    return createTransferMatrix(one, zero, zero, one);
}

void lmultm(TransferMatrix m, TransferMatrix mult) {
    complex A, B, C, D;

    A = addz(multz(mult->A, m->A), multz(mult->B, m->C));
    B = addz(multz(mult->A, m->B), multz(mult->B, m->D));
    C = addz(multz(mult->C, m->A), multz(mult->D, m->C));
    D = addz(multz(mult->C, m->B), multz(mult->D, m->D));

    m->A = A;
    m->B = B;
    m->C = C;
    m->D = D;
}

void rmultm(TransferMatrix m, TransferMatrix mult) {
    complex A, B, C, D;

    A = addz(multz(m->A, mult->A), multz(m->B, mult->C));
    B = addz(multz(m->A, mult->B), multz(m->B, mult->D));
    C = addz(multz(m->C, mult->A), multz(m->D, mult->C));
    D = addz(multz(m->C, mult->B), multz(m->D, mult->D));

    m->A = A;
    m->B = B;
    m->C = C;
    m->D = D;
}

complex calcZin(TransferMatrix m, complex Zload) {
    complex Zin, p1, p2, U1, U2;
}

```

```
p2 = (equalz(Zload, inf)) ?
    one :
    divz(Zload, addz(Zload, one));
U2 = (equalz(Zload, inf)) ?
    zero :
    divz(one, addz(Zload, one));
p1 = addz(multz(m->A, p2), multz(m->B, U2));
U1 = addz(multz(m->C, p2), multz(m->D, U2));
Zin = divz(p1, U1);
return Zin;
}

void invertm(TransferMatrix m) {
    complex det = subz(multz(m->A, m->D), multz(m->B, m->C));
    complex A = m->A;

    m->A = divz(m->D, det);
    m->B = divz(multz(real(-1.0), m->B), det);
    m->C = divz(multz(real(-1.0), m->C), det);
    m->D = divz(A, det);
}
```

Listing B.27: Vector.h

```

/*
Vector.h
By Andrew Botros, 2001-2003

Vector.c is a Vector ADT.
*/

#ifndef VECTOR_H_PROTECTOR
#define VECTOR_H_PROTECTOR

/* Node: { pointer to data struct, pointer to next Node } */
typedef struct Node_str {
    void* object;
    struct Node_str* next;
} *Node;

/* Vector:
   { size count, pointer to head Node, pointer to tail Node } */
typedef struct Root_str {
    int num;
    Node head;
    Node tail;
} *Vector;

Vector createVector(void);
/*
Initialises a vector with num = 0, head = NULL, tail = NULL.
Returns:
A Vector if successful, NULL otherwise.
*/
void addElement(Vector v, void* object);
/*
Adds a data structure pointer to the end of the given vector.
Parameters:
v: the Vector to add to.
object: a pointer to the data structure to be added.
*/
void insertAt(Vector v, void* object, int index);
/*
Inserts a data structure pointer within the given vector at the
given position.
All following pointers in the Vector are shifted right by index +1.
Parameters:
v: the Vector to add to.
object: a pointer to the data structure to be added.
index: the position of the inserted pointer (head is index 0).
NOTE: no bounds checking is performed.
*/
void setAt(Vector v, void* object, int index);
/*
Sets the data structure pointer within the given vector at the given

```

```

position.
Parameters:
  v: the Vector to be updated.
  object: a pointer to the data structure to be set.
  index: the position of the pointer to be altered (head is index
  0).
NOTE: no bounds checking is performed.
*/
void popFront(Vector v);
/*
  Removes the Node at the head of the Vector (index 0).
  If Vector is empty, nothing is done.
  All pointers in the Vector are shifted left by index -1.
Parameters:
  v: the Vector to be updated.
*/
void popBack(Vector v);
/*
  Removes the Node at the tail of the Vector.
  If Vector is empty, nothing is done.
Parameters:
  v: the Vector to be updated.
*/
void* elementAt(Vector v, int index);
/*
  Returns the data structure pointer within the given vector at the
  given index.
Parameters:
  v: the Vector to retrieve from.
  index: the position of the pointer requested (head is index 0).
Returns:
  The stored void* pointer within the Vector.
NOTE: no bounds checking is performed.
*/
int sizeVector(Vector v);
/*
  Returns the current number of data structure pointers stored within
  the given Vector.
Parameters:
  v: the given Vector.
Returns:
  The number of stored elements.
*/
#endif

```

Listing B.28: Vector.c

```

/*
Vector.c
By Andrew Botros, 2001-2003

Vector.c is a Vector ADT.
Refer to Vector.h for interface details.
*/

#include <stdio.h>
#include <stdlib.h>
#include "Vector.h"

Vector createVector(void) {
    /* allocate memory for *Vector */
    Vector v = (Vector)malloc(sizeof(*v));
    /* initialise num to 0, pointers to NULL */
    v->num = 0;
    v->head = NULL;
    v->tail = NULL;
    return v;
}

void addElement(Vector v, void* object) {
    /* if the Vector is empty, head and tail point to new object */
    if(v->tail == NULL) {
        v->tail = (Node)malloc(sizeof(*(v->tail)));
        v->tail->object = object;
        v->tail->next = NULL;
        v->head = v->tail;
    }
    /* else, add to end of Vector and update tail */
    else {
        v->tail->next = (Node)malloc(sizeof(*(v->tail->next)));
        v->tail->next->object = object;
        v->tail->next->next = NULL;
        v->tail = v->tail->next;
    }
    /* increment size */
    v->num++;
    return;
}

void insertAt(Vector v, void* object, int index) {
    int i;
    Node cur;
    Node prev;
    /* if Vector is empty and insert at 0, head and tail point to new
    object */
    if((v->num == 0) && (index == 0)) {
        v->tail = (Node)malloc(sizeof(*(v->tail)));
        v->tail->object = object;
        v->tail->next = NULL;
        v->head = v->tail;
    }
    else {

```

```

/* if inserted at beginning, update head */
if(index == 0) {
    cur = v->head;
    v->head = (Node)malloc(sizeof(*(v->head)));
    v->head->object = object;
    v->head->next = cur;
}
else {
    /* if inserted at end, update tail */
    if(index == v->num) {
        cur = v->tail;
        v->tail = (Node)malloc(sizeof(*(v->tail)));
        v->tail->object = object;
        v->tail->next = NULL;
        cur->next = v->tail;
    }
    /* otherwise, find position in Vector and update appropriately */
    else {
        cur = v->head->next;
        prev = v->head;
        for(i = 1; i < index; i++) {
            cur = cur->next;
            prev = prev->next;
        }
        prev->next = (Node)malloc(sizeof(*(prev->next)));
        prev->next->object = object;
        prev->next->next = cur;
    }
}
/* increment size */
v->num++;
return;
}

void setAt(Vector v, void* object, int index) {
    int i;
    Node cur;
    cur = v->head;
    /* find indexed element */
    for(i = 0; i < index; i++) {
        cur = cur->next;
    }
    /* update indexed element with given object */
    cur->object = object;
    return;
}

void popFront(Vector v) {
    Node second_element;
    /* do nothing if empty vector */
    if(sizeVector(v) == 0)
        return;
    /* if only 1 element, free head and reinitialise Vector */
    if(sizeVector(v) == 1) {
        free(v->head);
        v->head = NULL;
    }
}

```

```

v->tail = NULL;
v->num--;
return;
}
/* otherwise remove free head and update head pointer to second
element */
if(sizeVector(v) > 1) {
    second_element = v->head->next;
    free(v->head);
    v->head = second_element;
    v->num--;
    return;
}
}

void popBack(Vector v) {
    Node cur;
    /* do nothing if empty vector */
    if(sizeVector(v) == 0)
        return;
    /* if only 1 element, free tail and reinitialise Vector */
    if(sizeVector(v) == 1) {
        free(v->tail);
        v->head = NULL;
        v->tail = NULL;
        v->num--;
        return;
    }
    /* otherwise remove free tail and update tail pointer to second last
element */
    if(sizeVector(v) > 1) {
        /* advance cur to second last element */
        for(cur = v->head; cur->next != v->tail; cur = cur->next);
        free(v->tail);
        v->tail = cur;
        v->num--;
        return;
    }
}

void* elementAt(Vector v, int index) {
    int i;
    Node cur;
    cur = v->head;
    /* find indexed element */
    for(i = 0; i < index; i++) {
        cur = cur->next;
    }
    /* return found element */
    return cur->object;
}

int sizeVector(Vector v) {
    return v->num;
}

```

Listing B.29: Waves.c

```
/*
Waves.c
By Paul Dickens, 2005, 2006

Calculates the pressure and flow distribution along a flute.
*/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include "Acoustics.h"
#include "Woodwind.h"
#include "ParseXML.h"

int parseCommandLine(int argc, char** argv,
    char** holestring, double* xres, int* midi,
    double* f, char** xml_filename);

/* Default x resolution */
#define XRES 2.0e-3;

int main(int argc, char** argv) {
    double f;
    double x, xres, xmin, xmax;
    int midi;
    char* holestring;
    char* xml_filename;
    Woodwind instrument;
    complex Z0, Zin, pin, Uin;
    double c;
    TransferMatrix m;
    complex p, U;
    double entryradius = WW_EMB_RADIUS;

    /* check correct usage */
    if(!parseCommandLine(argc, argv, &holestring, &xres, &midi,
        &f, &xml_filename)) {
        fprintf(stderr,
            "Usage: Waves [OPTIONS] <midi> <frequency> <XML file>\n\n");
        fprintf(stderr, " Options:\n");
        fprintf(stderr, "\t-s <holestring>\n");
        fprintf(stderr, "\t-r <xres> (default 2.0)\n\n");
        fprintf(stderr, " <holestring>:\n");
        fprintf(stderr,
            "\t- Optional if no holes are defined in XML file.\n");
        fprintf(stderr, "\t- Must be a sequence of '0' (open hole) ");
        fprintf(stderr, "and 'X' (closed hole) characters.\n");
        fprintf(stderr,
            "\t- Length must be equal to the number of defined holes.\n");
        fprintf(stderr, "\t- e.g. \"XX000000X0000X000\"\n\n");
        return -1;
    }

    /* retrieve data structures from XML file */
    if(!parseXMLFile(xml_filename, &instrument)) {
```

```

        fprintf(stderr, "Waves error: Waves failed to parse XML file.\n");
        return -1;
    }

discretiseWoodwind(instrument, WW_MAX_LENGTH);
setAirProperties(instrument, WW_T_0, WW_T_AMB, WW_T_GRAD, WW_HUMID,
WW_X_CO2);

/* set fingering from holestring and validate */
if(!setFingering(instrument, holestring)) {
    fprintf(stderr, "Waves error: \"%s\" ", holestring);
    fprintf(stderr,
            "is an invalid fingering for the given woodwind definition.\n");
    return -1;
}

/* calculate Zin, c and Z0 */
Zin = playedImpedance(f, instrument, midi);
c = instrument->head->embouchureHole->c;
Z0 = charZ(c, instrument->head->embouchureHole->rho,
entryradius);

/* calculate pin and Uin */
Uin = real(1 / sqrt(modz(Zin)*modz(Zin) + modz(Z0)*modz(Z0)));
// Uin = real(sqrt(2 * modz(Z0) * c / (modz(Z0)*modz(Z0)
//      + modz(Zin)*modz(Zin))));

pin = multz(Zin, Uin);

/* change pin to account for face impedance */
pin = subz(pin, multz(faceZ(f, instrument->head, midi), Uin));

/* calculate the limits of the instrument */
xmin = ceil(-woodwindLengthNeg(instrument) / xres) * xres;
xmax = ceil(woodwindLengthPos(instrument) / xres) * xres;
for (x = xmin; x < xmax; x+=xres) {
    m = woodwindMatrix(f, instrument,
        entryradius / woodwindEntryRadius(instrument), x);
    invertm(m);
    p = addz(multz(m->A, pin), multz(m->B, Uin));
    U = addz(multz(m->C, pin), multz(m->D, Uin));
    // getZ0_c(instrument, x, &Z0, &c);
    printf("%.1f\t%.3f\t%.3f\n", x * 1e3, modz(p), modz(Z0)*modz(U));
}
return 0;
}

int parseCommandLine(int argc, char** argv,
char** holestring, double* xres, int* midi,
double* f, char** xml_filename) {
int i;
int sflag = 0, rflag = 0;
int numoptions = 2, numrequired = 3;
int minargc = 1 + numrequired;
int maxargc = minargc + 2*numoptions;

/* Check correct number of parameters */
if((argc < minargc) || (argc%2 != minargc%2) || (argc > maxargc))

```

```
return 0;

/* Set default options */
*holestring = NULL;
*xres = XRES;

/* Check and set options */
for(i = 1; i < (argc - numrequired); i += 2) {
    if(strcmp(argv[i], "-s") == 0) {
        if(sflag)
            return 0;
        *holestring = argv[i + 1];
        sflag = 1;
        continue;
    }
    if(strcmp(argv[i], "-r") == 0) {
        if(rflag)
            return 0;
        *xres = atof(argv[i + 1]);
        if(*xres <= 0.0) {
            fprintf(stderr, "Invalid -r option\n");
            return 0;
        }
        rflag = 1;
        continue;
    }
    /* else invalid option */
    fprintf(stderr, "Invalid option: %s\n", argv[i]);
    return 0;
}
/* Set midi */
*midi = atof(argv[argc - numrequired]);
/* Set f */
*f = atof(argv[argc - numrequired + 1]);
if(*f <= 0.0) {
    fprintf(stderr, "Invalid frequency\n");
    return 0;
}
/* Set XML filename */
*xml_filename = argv[argc - numrequired + 2];
return 1;
}
```

Listing B.30: Woodwind.h

```

/*
Woodwind.h
By Paul Dickens, 2005

A woodwind instrument.

*/


---


#ifndef WOODWIND_H_PROTECTOR
#define WOODWIND_H_PROTECTOR

#include "Vector.h"
#include "Map.h"
#include "Complex.h"
#include "TransferMatrix.h"

/* Maximum length of bore elements */
#define WW_MAX_LENGTH 5.0e-3

/* Temperature, humidity and CO2 */
#define WW_T_0 30.3
#define WW_T_AMB 21.0
#define WW_T_GRAD -7.7 // degrees C per metre
#define WW_HUMID 1
#define WW_X_CO2 0.025

/* Embouchure entry radius */
#define WW_EMB_RADIUS 3.9e-3;

/* Empirical corrections */
#define CORR_OPEN_FINGER_HOLE_LENGTH -0.15
#define CORR_OPEN_KEYED_HOLE_LENGTH 0.1
#define CORR_CLOSED_FINGER_HOLE_LENGTH -0.76
#define CORR_CLOSED_KEYED_HOLE_LENGTH -0.05

/* BoreSegment: */
typedef struct boresegment_str {
    double radius1;
    double radius2;
    double length;
    double c;
    double rho;
} *BoreSegment;

/* Key: */
typedef struct key_str {
    double radius;
    double holeRadius;
    double height;
    double thickness;
    double wallThickness;
    double chimneyHeight;
} *Key;

```

```

/* Hole: */
typedef struct hole_str {
    double radius;
    double length;
    double boreRadius;
    Key key;
    double c;
    double rho;
    char* fingering;
} *Hole;

/* EmbouchureHole: */
typedef struct embouchurehole_str {
    double radiusin;
    double radiusout;
    double length;
    double boreRadius;
    double c;
    double rho;
} *EmbouchureHole;

/* Head: */
typedef struct head_str {
    EmbouchureHole embouchureHole;
    Vector upstreamBore;
    double upstreamFlange;
    Vector downstreamBore;
    Map matrixMap;
} *Head;

/* UnitCell: */
typedef struct unitcell_str {
    Hole hole;
    Vector bore;
    Map openMatrixMap;
    Map closedMatrixMap;
} *UnitCell;

/* Woodwind: */
typedef struct woodwind_str {
    Head head;
    Vector cells;
    double flange;
} *Woodwind;

BoreSegment createBoreSegment(double radius1, double radius2,
    double length);
/*
Creates a new BoreSegment.
Parameters:
    radius1: input radius in m
    radius2: output radius in m
    length: segment length in m
Returns:
    a new BoreSegment with the given parameters
*/
Hole createHole(double radius, double length, double boreRadius,

```

```

    Key key);
/*
Creates a new Hole.
Parameters:
    radius: hole radius in m
    length: segment length in m
    boreRadius: the radius of the instrument bore at the position of
    the hole
Returns:
    a new Hole with the given parameters
*/
Key createKey(double radius, double boreRadius, double height,
    double thickness, double wallThickness, double chimneyHeight);
/*
Creates a new Key.
Parameters:
    radius: the radius of the key in m
    holeRadius: the radius of the hole in the key (for perforated
    keys)
    height: the regulation height of the open key
    thickness: the thickness of the key
    wallThickness: the thickness of the hole chimney
    chimneyHeight: the height of the chimney above the bore
Returns:
    a new Key with the given parameters
*/
EmbouchureHole createEmbouchureHole(double radiusin, double radiusout,
    double length, double boreRadius);
/*
Creates a new EmbouchureHole.
Parameters:
    radiusin: the radius at the bore
    radiusout: the radius at the outside
    length: the embouchure hole length
    boreRadius: the radius of the instrument bore at the position of
    the hole
Returns:
    a new EmbouchureHole with the given parameters
*/
Head createHead(EmbouchureHole embouchureHole, Vector upstreamBore,
    double upstreamFlange, Vector downstreamBore);
/*
Creates a new Head.
Parameters:
    embouchureHole: the embouchure hole
    upstreamBore: bore section between the embouchure hole and the
    cork
    upstreamFlange: the termination condition at the cork (default
    -1)
    downstreamBore: bore section between the embouchure hole and the
    first tone hole
Returns:
    a new Head with the given parameters
*/

```

```

UnitCell createUnitCell(Hole hole, Vector bore);
/*
Creates a new UnitCell.
Parameters:
hole: the tone hole
bore: the bore section downstream of the hole (optional)
Returns:
a new UnitCell with the given parameters
*/
Woodwind createWoodwind(Head head, Vector cells, double flange);
/*
Creates a new Woodwind.
Parameters:
head: the Head of the woodwind
cells: a vector of unit cells
flange: the termination condition at the end of the instrument
Returns:
a new Woodwind with the given parameters
*/
void setAirProperties(Woodwind w, double t_0, double t_amb,
double t_grad, double humid, double x_CO2);
/*
Sets the speed of sound and density of air along the instrument.
Parameters:
w: the instrument
t_0: the temperature at x = 0 (embouchure hole) in deg C
t_amb: the ambient temperature (deg C)
t_grad: the temperature gradient (deg C / m)
humid: the relative humidity (between 0 and 1)
x_CO2: the molar fraction of carbon dioxide
*/
void discretiseWoodwind(Woodwind w, double maxLength);
/*
Cuts up instrument so that no segment is longer than maxLength.
Parameters:
w: the instrument
maxLength: the maximum segment length
*/
void discretiseBore(Vector bore, double maxLength);
/*
Cuts up a bore so that no segment is longer than maxLength.
Parameters:
w: the instrument
maxLength: the maximum segment length
*/
int setFingering(Woodwind w, char* holestring);
/*
Sets the woodwind fingering to the given string.
Parameters:
w: the instrument
holestring: a string of 'X's and 'O's representing the fingering
Returns:
1 if the operation was sucessful

```

```

    0 otherwise
*/
TransferMatrix boreSegmentMatrix(double f, BoreSegment s, double x);
/*
   Calculates the TransferMatrix for a BoreSegment.
   Parameters:
      f: the frequency in Hz
      s: the BoreSegment
      x: distance along the BoreSegment to calculate
   Returns:
      the TransferMatrix for the BoreSegment
*/
TransferMatrix boreMatrix(double f, Vector bore, double x);
/*
   Calculates the TransferMatrix for a bore.
   Parameters:
      f: the frequency in Hz
      bore: the bore
      x: distance along the bore to calculate
   Returns:
      the TransferMatrix for the bore
*/
TransferMatrix headMatrix(double f, Head h, double entryratio,
                         double x);
/*
   Calculates the TransferMatrix for a Head.
   Parameters:
      f: the frequency in Hz
      h: the Head
      entryratio: the ratio of the input side radius (impedance head or
                  embouchure) to the entry radius of the instrument
      x: distance along the Head to calculate
   Returns:
      the TransferMatrix for the Head
*/
complex faceZ(double f, Head head, int midi);
/*
   Calculates the radiation impeance of the player's face.
   Parameters:
      f: the frequency in Hz
      head: the Head
      midi: the MIDI number for the played note
   Returns:
      the radiation impedance
*/
TransferMatrix unitCellMatrix(double f, UnitCell c, double x);
/*
   Calculates the TransferMatrix for a UnitCell.
   Parameters:
      f: the frequency in Hz
      c: the UnitCell
      x: distance along the UnitCell to calculate
   Returns:

```

```

    the TransferMatrix for the UnitCell
*/
TransferMatrix woodwindMatrix(double f, Woodwind w, double entryratio,
    double x);
/*
    Calculates the TransferMatrix for a Woodwind.
Parameters:
    f: the frequency in Hz
    w: the Woodwind
    entryratio: the ratio of the input side radius (impedance head or
        embouchure) to the entry radius of the instrument
    x: distance along the Woodwind to calculate
Returns:
    the TransferMatrix for the Woodwind
*/
int getZ0_c(Woodwind w, double x, complex* Z0, double* c);
/*
    Calculates the characteristic impedance and speed of sound at a
    given lateral position along an instrument.
Parameters:
    w: the Woodwind
    x: the distance along the Woodwind
    Z0: the return variable for the characteristic impedance
    c: the return variable for the speed of sound
Returns:
    1 if x within range
    0 otherwise
*/
double boreLength(Vector bore);
/*
    Calculates the length of a bore.
Parameters:
    bore: a vector of BoreSegments
Returns:
    the total length of the bore
*/
double woodwindLengthPos(Woodwind w);
/*
    Calculates the length of a Woodwind in the positive direction.
Parameters:
    w: the Woodwind
Returns:
    the length of the Woodwind from the embouchure hole in the
        positive direction
*/
double woodwindLengthNeg(Woodwind w);
/*
    Calculates the length of a Woodwind in the negative direction.
Parameters:
    w: the Woodwind
Returns:
    the length of the Woodwind from the embouchure hole in the
        negative direction

```

```

*/
complex impedance(double f, Woodwind w, double entryratio);
/*
Calculates the input impedance of a Woodwind.
Parameters:
f: the frequency in Hz
w: the Woodwind
entryratio: the ratio of the input side radius (impedance head or
embouchure) to the entry radius of the instrument
Returns:
the input impedance of the woodwind
*/
complex playedImpedance(double f, Woodwind w, int midi);
/*
Calculates the input impedance of a Woodwind in combination with the
player's face impedance.
Parameters:
f: the frequency in Hz
w: the Woodwind
midi: the MIDI number of the played note
Returns:
the input impedance of the played woodwind
*/
double woodwindEntryRadius(Woodwind w);
/*
Calculates the radius of the woodwind at the entry.
Parameters:
w: the Woodwind
Returns:
the outside radius of the embouchure hole (if present)
otherwise the input radius of the first BoreSegment
*/
complex woodwindLoadZ(double f, Woodwind w);
/*
Calculates the load impedance at the end of a Woodwind.
Parameters:
f: the frequency in Hz
w: the Woodwind
Returns:
the load impedance of the woodwind
*/
complex woodwindDownstreamZ(double f, Woodwind w);
/*
Calculates the input impedance of the section of the Woodwind
downstream of the embouchure hole.
Parameters:
f: the frequency in Hz
w: the Woodwind
Returns:
the downstream impedance of the Woodwind
*/
TransferMatrix traverseHoleMatrix(double f, Hole h);

```

```

/*
  Calculates the transfer matrix relating the acoustic parameters on
  each side of the hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The transfer matrix for the hole.
*/

complex holeInputImpedance(double f, Hole hole);
/*
  Calculates the input impedance of a Hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The input impedance of the hole.
*/

complex closedFingerHoleLoadZ(double f, Hole hole);
/*
  Calculates the load impedance of a closed finger hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The load impedance of the hole.
*/

complex closedKeyedHoleLoadZ(double f, Hole hole);
/*
  Calculates the load impedance of a closed keyed hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The load impedance of the hole.
*/

complex openFingerHoleLoadZ(double f, Hole hole);
/*
  Calculates the load (radiation) impedance of an open finger hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The load (radiation) impedance of the hole.
*/

complex openKeyedHoleLoadZ(double f, Hole hole);
/*
  Calculates the load (radiation) impedance of an open keyed hole.
  Parameters:
    f: the frequency.
    h: the hole.
  Returns:
    The load (radiation) impedance of the hole.
*/

```

```

*/
complex holeInnerRadiationImpedance(double f, Hole hole);
/*
Calculates the inner radiation impedance of a hole.
Parameters:
f: the frequency.
h: the hole.
Returns:
The inner radiation impedance of the hole.
*/
complex holeSeriesImpedance(double f, Hole hole);
/*
Calculates the series impedance of a hole.
Parameters:
f: the frequency.
h: the hole.
Returns:
The series impedance of the hole.
*/
TransferMatrix embouchureMatrix(double f, EmbouchureHole h,
double entryratio, complex branchZ);
/*
Calculates the TransferMatrix for an EmbouchureHole.
Parameters:
f: the frequency in Hz
h: the EmbouchureHole
entryratio: the ratio of the input side radius (impedance head or
embouchure) to the outside radius of the embouchure hole
branchZ: the impedance of the impedance branch (upstream or
downstream section)
Returns:
the TransferMatrix relating the acoustic parameters on the input
side of the embouchure hole to those at the output side (inside of
the instrument)
*/
double embouchureLengthCorrection(double a, double b);
/*
Returns the empirically-determined embouchure length correction.
Parameters:
a: the bore radius at the embouchure hole
b: the inside radius of the embouchure hole
Returns:
the length correction
*/
complex embouchureSeriesResistance(double f, EmbouchureHole h,
double entryratio);
/*
Returns the empirically-determined embouchure series resistance
correction.
Parameters:
f: the frequency in Hz
h: the EmbouchureHole
entryratio: the ratio of the input side radius (impedance head or

```

```

embouchure) to the outside radius of the embouchure hole
branchZ: the impedance of the impedance branch (upstream or
downstream section)
>Returns:
the series resistance
*/
complex embouchureShuntConductance(double f, EmbouchureHole h,
double entryratio);
/*
Returns the empirically-determined embouchure shunt conductance
correction.
Parameters:
f: the frequency in Hz
h: the EmbouchureHole
entryratio: the ratio of the input side radius (impedance head or
embouchure) to the outside radius of the embouchure hole
branchZ: the impedance of the impedance branch (upstream or
downstream section)
>Returns:
the shunt conductance
*/
double matchingLengthCorrection(double a, double b);
/*
Returns the length correction corresponding to the 'matching volume'
at the junction between a bore and a hole.
Parameters:
a: the bore radius at the hole
b: the hole radius
>Returns:
the matching length correction
*/
double innerRadiationLengthCorrection(double a, double b);
/*
Returns the length correction corresponding to inner radiation at
the junction between a bore and a hole.
Parameters:
a: the bore radius at the hole
b: the hole radius
>Returns:
the inner radiation length correction
*/
double closedHoleSeriesLengthCorrection(double a, double b, double t);
/*
Returns the length correction corresponding to flow widening at the
junction between a bore and a closed hole.
Parameters:
a: the bore radius at the hole
b: the hole radius
>Returns:
the series length correction
*/
double openHoleSeriesLengthCorrection(double a, double b);
/*

```

Returns the length correction corresponding to flow widening at the junction between a bore and an open hole.

Parameters:

a: the bore radius at the hole

b: the hole radius

Returns:

the series length correction

**/*

#endif

Listing B.31: Woodwind.c

```
/*
Woodwind.c
By Paul Dickens, 2005, 2006.

A woodwind instrument. See Woodwind.h for interface details.
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "Woodwind.h"
#include "Acoustics.h"
#include "Map.h"

#define P_ATM 101325 // 1 atm pressure

BoreSegment createBoreSegment(double radius1, double radius2,
    double length) {
    BoreSegment s = (BoreSegment)malloc(sizeof(*s));
    s->radius1 = radius1;
    s->radius2 = radius2;
    s->length = length;
    s->c = 0.0;
    s->rho = 0.0;
    return s;
}

Hole createHole(double radius, double length, double boreRadius,
    Key key) {
    Hole h = (Hole)malloc(sizeof(*h));
    h->radius = radius;
    h->length = length;
    h->boreRadius = boreRadius;
    h->key = key;
    h->c = 0.0;
    h->rho = 0.0;
    return h;
}

Key createKey(double radius, double holeRadius, double height,
    double thickness, double wallThickness, double chimneyHeight) {
    Key k = (Key)malloc(sizeof(*k));
    k->radius = radius;
    k->holeRadius = holeRadius;
    k->height = height;
    k->thickness = thickness;
    k->wallThickness = wallThickness;
    k->chimneyHeight = chimneyHeight;
    return k;
}

EmbouchureHole createEmbouchureHole(double radiusin, double radiusout,
```

```

        double length, double boreRadius) {
EmbouchureHole h = (EmbouchureHole)malloc(sizeof(*h));
h->radiusin = radiusin;
h->radiusout = radiusout;
h->length = length;
h->boreRadius = boreRadius;
h->c = 0.0;
h->rho = 0.0;
return h;
}

Head createHead(EmbouchureHole embouchureHole, Vector upstreamBore,
    double upstreamFlange, Vector downstreamBore) {
Head h = (Head)malloc(sizeof(*h));
h->embouchureHole = embouchureHole;
h->upstreamBore = upstreamBore;
h->upstreamFlange = upstreamFlange;
h->downstreamBore = downstreamBore;
h->matrixMap = createMap();
return h;
}

UnitCell createUnitCell(Hole hole, Vector bore) {
UnitCell c = (UnitCell)malloc(sizeof(*c));
c->hole = hole;
c->bore = bore;
c->openMatrixMap = createMap();
c->closedMatrixMap = createMap();
return c;
}

Woodwind createWoodwind(Head head, Vector cells, double flange) {
Woodwind w = (Woodwind)malloc(sizeof(*w));
w->head = head;
w->cells = cells;
w->flange = flange;
return w;
}

void setAirProperties(Woodwind w, double t_0, double t_amb,
    double t_grad, double humid, double x_CO2) {
Head h = w->head;
double temp, x;
int segmentCount, cellCount;
BoreSegment s;
UnitCell cell;

temp = t_0;
if (w->head->embouchureHole != NULL) {
/* set c and rho for the embouchure hole */
h->embouchureHole->c = speedSound(temp, P_ATM, humid, x_CO2);
h->embouchureHole->rho =
    densityAirGiacomo(temp, P_ATM, humid, x_CO2);

x = 0;
/* for each bore segment in upstream */
for (segmentCount = 0; segmentCount < sizeVector(h->upstreamBore);
    segmentCount++) {

```

```

s = (BoreSegment)elementAt(h->upstreamBore, segmentCount);
temp = t_0 + t_grad * (x + s->length / 2);
if (temp < t_amb) temp = t_amb;
s->c = speedSound(temp, P_ATM, humid, x_CO2);
s->rho = densityAirGiacomo(temp, P_ATM, humid, x_CO2);
x += s->length;
}
}
x = 0;
/* for each bore segment in downstream */
for (segmentCount = 0; segmentCount < sizeVector(h->downstreamBore);
    segmentCount++) {
    s = (BoreSegment)elementAt(h->downstreamBore, segmentCount);
    temp = t_0 + t_grad * (x + s->length / 2);
    if (temp < t_amb) temp = t_amb;
    s->c = speedSound(temp, P_ATM, humid, x_CO2);
    s->rho = densityAirGiacomo(temp, P_ATM, humid, x_CO2);
    x += s->length;
}

/* for each unit cell */
for (cellCount = 0; cellCount < sizeVector(w->cells); cellCount++) {
    cell = (UnitCell)elementAt(w->cells, cellCount);
    temp = t_0 + t_grad * x;
    if (temp < t_amb) temp = t_amb;
    cell->hole->c = speedSound(temp, P_ATM, humid, x_CO2);
    cell->hole->rho = densityAirGiacomo(temp, P_ATM, humid, x_CO2);
    /* for each bore segment in unit cell */
    for (segmentCount = 0; segmentCount < sizeVector(cell->bore);
        segmentCount++) {
        s = (BoreSegment)elementAt(cell->bore, segmentCount);
        temp = t_0 + t_grad * (x + s->length / 2);
        if (temp < t_amb) temp = t_amb;
        s->c = speedSound(temp, P_ATM, humid, x_CO2);
        s->rho = densityAirGiacomo(temp, P_ATM, humid, x_CO2);
        x += s->length;
    }
}
}

void discretiseWoodwind(Woodwind w, double maxLength) {
    int cellCount;
    UnitCell cell;

    discretiseBore(w->head->upstreamBore, maxLength);
    discretiseBore(w->head->downstreamBore, maxLength);
    for (cellCount = 0; cellCount < sizeVector(w->cells); cellCount++) {
        cell = (UnitCell)elementAt(w->cells, cellCount);
        discretiseBore(cell->bore, maxLength);
    }
}

void discretiseBore(Vector bore, double maxLength) {
    int segmentCount, newSegmentCount, numSegments;
    BoreSegment s, newSegment;
    double radius1, radius2, length;

    /* for each bore segment in bore */

```

```

for (segmentCount = 0; segmentCount < sizeVector(bore);
    segmentCount++) {
    s = (BoreSegment)elementAt(bore, segmentCount);
    if (s->length <= maxLength) continue;
    numSegments = ceil(s->length / maxLength);
    radius1 = s->radius1;
    length = s->length / numSegments;
    for (newSegmentCount = 1; newSegmentCount <= numSegments;
        newSegmentCount++) {
        radius2 = (newSegmentCount * s->radius2
            + (numSegments - newSegmentCount) * s->radius1) / numSegments;
        newSegment = createBoreSegment(radius1, radius2, length);
        if (newSegmentCount == 1)
            setAt(bore, newSegment, segmentCount);
        else
            insertAt(bore, newSegment, ++segmentCount);
        radius1 = radius2;
    }
}
}

int setFingering(Woodwind w, char* holestring) {
    int numholes = sizeVector(w->cells);
    int i;
    UnitCell cell;
    /* check for no holes */
    if((holestring == NULL) || (numholes == 0)) {
        if((holestring == NULL) && (numholes == 0))
            return 1;
        else
            return 0;
    }
    /* check for wrong number of holes entered */
    if(strlen(holestring) != numholes)
        return 0;
    /* set fingering array based on command line hole string */
    for(i = 0; i < numholes; i++) {
        cell = (UnitCell)elementAt(w->cells, i);
        if(holestring[i] == 'O')
            cell->hole->fingering = "OPEN";
        if(holestring[i] == 'X')
            cell->hole->fingering = "CLOSED";
        if((holestring[i] != 'O') && (holestring[i] != 'X'))
            return 0;
    }
    return 1;
}

TransferMatrix boreSegmentMatrix(double f, BoreSegment s, double x) {
    TransferMatrix m;
    double length, radius1, radius2;

    radius1 = s->radius1;
    if (x >= s->length) {
        length = s->length;
        radius2 = s->radius2;
    }
    else {

```

```

length = x;
radius2 = (s->radius2 * length + s->radius1
           * (s->length - length)) / (s->length);
}
m = (radius1 == radius2) ?
    tubeMatrix(f, s->c, s->rho, length, radius1, 1) :
    coneMatrix(f, s->c, s->rho, length, radius1, radius2, 1);
return m;
}

TransferMatrix boreMatrix(double f, Vector bore, double x) {
    TransferMatrix m = identitym();
    int n = 0;
    BoreSegment s;

    while ((x > 0) && (n < sizeVector(bore))) {
        s = (BoreSegment)elementAt(bore, n);
        rmultm(m, boreSegmentMatrix(f, s, x));
        x -= s->length;
        n++;
    }
    return m;
}

TransferMatrix headMatrix(double f, Head h, double entryratio,
                         double x) {
    TransferMatrix m, branchMatrix;
    complex branchZ, ZL;
    BoreSegment lastSegment;
    Map map = h->matrixMap;

    if((x >= boreLength(h->downstreamBore)) && containsKey(map, f))
        m = (TransferMatrix)get(map,f);
    else {
        m = identitym();
        if (h->embouchureHole != NULL) {
            branchMatrix =
                boreMatrix(f, h->upstreamBore, boreLength(h->upstreamBore));
            lastSegment = (BoreSegment)elementAt(h->upstreamBore,
                                                 sizeVector(h->upstreamBore) - 1);
            ZL = radiationZ(f, lastSegment->c, lastSegment->rho,
                            lastSegment->radius2, h->upstreamFlange);
            branchZ = calcZin(branchMatrix, ZL);
            rmultm(m,
                   embouchureMatrix(f, h->embouchureHole, entryratio, branchZ));
        }
        if (x > 0)
            rmultm(m, boreMatrix(f, h->downstreamBore, x));
        if (x >= boreLength(h->downstreamBore)) {
            clear(map);
            put(map, f, m);
        }
    }
    return m;
}

complex faceZ(double f, Head head, int midi) {
    double c = head->embouchureHole->c;
}

```

```

double rho = head->embouchureHole->rho;
double entryradius = WW_EMB_RADIUS;
double corr = 2.9370 * log(midi) - 11.6284;
return multz(flangedZ(f, c, rho, entryradius), real(corr));
}

TransferMatrix unitCellMatrix(double f, UnitCell c, double x) {
    TransferMatrix m;
    Map map = (strcmp(c->hole->fingering, "OPEN") == 0) ?
        c->openMatrixMap :
        c->closedMatrixMap;

    if((x >= boreLength(c->bore)) && containsKey(map, f))
        m = (TransferMatrix)get(map, f);
    else {
        m = traverseHoleMatrix(f, c->hole);
        if((x > 0) && (c->bore != NULL))
            rmultm(m, boreMatrix(f, c->bore, x));
        if (x >= boreLength(c->bore)) {
            clear(map);
            put(map, f, m);
        }
    }
    return m;
}

TransferMatrix woodwindMatrix(double f, Woodwind w, double entryratio,
    double x) {
    TransferMatrix m = identitym();
    Head h;
    complex branchZ;
    UnitCell cell;
    int cellCount = 0;

    if (x >= 0) {
        rmultm(m, headMatrix(f, w->head, entryratio, x));
        x -= boreLength(w->head->downstreamBore);
        while (x > 0 && cellCount < sizeVector(w->cells)) {
            cell = (UnitCell)elementAt(w->cells, cellCount);
            rmultm(m, unitCellMatrix(f, cell, x));
            x -= boreLength(cell->bore);
            cellCount++;
        }
    }
    else {
        h = w->head;
        if (h->embouchureHole != NULL) {
            branchZ = woodwindDownstreamZ(f, w);
            rmultm(m, embouchureMatrix(f, h->embouchureHole, entryratio,
                branchZ));
        }
        rmultm(m, boreMatrix(f, h->upstreamBore, -x));
    }
    return m;
}

int getZ0_c(Woodwind w, double x, complex* Z0, double* c) {
    Vector bore;

```

```

int n = 0, cellCount = 0;
UnitCell cell;
BoreSegment s;
double radius;

if(x == 0) {
    *Z0 = charZ(w->head->embouchureHole->c,
    w->head->embouchureHole->rho,
    w->head->embouchureHole->boreRadius);
    *c = w->head->embouchureHole->c;
    return 1;
}
if (x > 0) {
    /* find the bore or hole at the point x */
    bore = w->head->downstreamBore;
    /* if x is outside of the current bore, try the next one */
    while(x >= boreLength(bore) && cellCount < sizeVector(w->cells)) {
        /* subtract from x the length of the previous bore */
        x -= boreLength(bore);
        /* find the next bore */
        cell = (UnitCell)elementAt(w->cells, cellCount++);
        bore = cell->bore;
    }
}
else if (x < 0) {
    bore = w->head->upstreamBore;
    x = -x;
}
/* if we are at a hole */
if(x == 0) {
    *Z0 = charZ(cell->hole->c, cell->hole->rho,
    cell->hole->boreRadius);
    *c = cell->hole->c;
    return 1;
}
/* otherwise, find the correct BoreSegment */
else {
    s = (BoreSegment)elementAt(bore, n);
    while(x > s->length) {
        /* subtract from x the length of the previous segment */
        x -= s->length;
        /* get the next segment */
        s = (BoreSegment)elementAt(bore, ++n);
    }
}
/* if x is outside the BoreSegment, return 0 */
if(x > s->length) return 0;
/* otherwise do a linear interpolation */
else {
    radius = (s->radius2 * x + s->radius1 * (s->length - x))
    / (s->length);
}
*Z0 = charZ(s->c, s->rho, radius);
*c = s->c;
return 1;
}

double boreLength(Vector bore) {

```

```

int n;
BoreSegment s;
double length = 0;

for(n = 0; n < sizeVector(bore); n++) {
    s = (BoreSegment)elementAt(bore, n);
    length += s->length;
}
return length;
}

double woodwindLengthPos(Woodwind w) {
    int cellCount;
    UnitCell cell;
    double length = 0;

    length += boreLength(w->head->downstreamBore);
    for (cellCount = 0; cellCount < sizeVector(w->cells); cellCount++) {
        cell = (UnitCell)elementAt(w->cells, cellCount);
        length += boreLength(cell->bore);
    }
    return length;
}

double woodwindLengthNeg(Woodwind w) {
    return boreLength(w->head->upstreamBore);
}

complex impedance(double f, Woodwind w, double entryratio) {
    TransferMatrix matrix =
        woodwindMatrix(f, w, entryratio, woodwindLengthPos(w));
    return calcZin(matrix, woodwindLoadZ(f, w));
}

complex playedImpedance(double f, Woodwind w, int midi) {
    double entryradius = WW_EMB_RADIUS;
    complex Z = impedance(f, w, entryradius / woodwindEntryRadius(w));
    Z = addz(Z, faceZ(f, w->head, midi));
    return Z;
}

double woodwindEntryRadius(Woodwind w) {
    double a;
    BoreSegment s;

    if (w->head->embouchureHole != NULL)
        a = w->head->embouchureHole->radiusout;
    else {
        s = (BoreSegment)elementAt(w->head->downstreamBore, 0);
        a = s->radius1;
    }
    return a;
}

complex woodwindLoadZ(double f, Woodwind w) {
    Vector lastBore;
    UnitCell lastCell;
    BoreSegment s;
}

```

```

if (sizeVector(w->cells) == 0)
    lastBore = w->head->downstreamBore;
else {
    lastCell = (UnitCell)elementAt(w->cells,
        sizeVector(w->cells) - 1);
    lastBore = lastCell->bore;
}
s = (BoreSegment)elementAt(lastBore, sizeVector(lastBore) - 1);
return radiationZ(f, s->c, s->rho, s->radius2, w->flange);
}

complex woodwindDownstreamZ(double f, Woodwind w) {
    TransferMatrix m;
    UnitCell cell;
    int cellCount;

    m = boreMatrix(f, w->head->downstreamBore,
        boreLength(w->head->downstreamBore));
    for (cellCount = 0; cellCount < sizeVector(w->cells); cellCount++) {
        cell = (UnitCell)elementAt(w->cells, cellCount);
        rmultm(m, unitCellMatrix(f, cell, boreLength(cell->bore)));
    }
    return calcZin(m, woodwindLoadZ(f, w));
}

TransferMatrix traverseHoleMatrix(double f, Hole hole) {
    TransferMatrix m = identitym();
    complex Z_hole, Z_i, Z_a;

    /* calculate the input impedance to the hole */
    Z_hole = holeInputImpedance(f, hole);

    /* calculate the inner radiation impedance */
    Z_i = holeInnerRadiationImpedance(f, hole);

    /* calculate the series impedance */
    Z_a = holeSeriesImpedance(f, hole);

    /* assign the impedances to the correct matrix element */
    m->C = divz(one, addz(Z_i, Z_hole));
    m->B = Z_a;

    return m;
}

complex holeInputImpedance(double f, Hole hole) {
    TransferMatrix holeMatrix;
    double t;
    double a = hole->boreRadius;
    double b = hole->radius;
    complex Z_L;

    /* calculate the length t (including the matching length
       correction) */
    t = hole->length + matchingLengthCorrection(a, b);

    /* calculate the matrix (with losses) for the tube section

```

```

comprising the hole and matching length */
holeMatrix = tubeMatrix(f, hole->c, hole->rho, t, b, 1);

/* calculate the load (radiation) impedance of the hole */
if (strcmp(hole->fingering, "CLOSED") == 0)
    Z_L = (hole->key == NULL) ?
        closedFingerHoleLoadZ(f, hole) :
        closedKeyedHoleLoadZ(f, hole);
else
    Z_L = (hole->key == NULL) ?
        openFingerHoleLoadZ(f, hole) :
        openKeyedHoleLoadZ(f, hole);

return calcZin(holeMatrix, Z_L);
}

complex closedFingerHoleLoadZ(double f, Hole hole) {
    double a = hole->boreRadius;
    double b = hole->radius;
    double delta = b / a;
    double t_finger;
    double Z0 = charZ(hole->c, hole->rho, hole->radius).Re;
    double k = (2*M_PI*f)/hole->c;

    t_finger = CORR_CLOSED_FINGER_HOLE_LENGTH * delta * b;

    return imaginary(-Z0 / tan(k * t_finger));
}

complex closedKeyedHoleLoadZ(double f, Hole hole) {
    double b = hole->radius;
    double t_keypad;
    double Z0 = charZ(hole->c, hole->rho, hole->radius).Re;
    double k = (2*M_PI*f)/hole->c;

    t_keypad = CORR_CLOSED_KEYED_HOLE_LENGTH * b;
    if (t_keypad == 0) return inf;
    return imaginary(-Z0 / tan(k * t_keypad));
}

complex openFingerHoleLoadZ(double f, Hole hole) {
    double a, b, k;
    complex Z_flanged, Z0, d_flanged, d_cyl, Z;

/* a is the radius of the hole */
a = hole->radius;

/* b is the radius of the (cylindrical) flange */
b = hole->boreRadius + hole->length;

/* calculate the wavenumber */
k = (2*M_PI*f)/hole->c;

/* calculate the characteristic impedance */
Z0 = charZ(hole->c, hole->rho, a);

/* calculate the impedance of the (infinitely) flanged hole */
Z_flanged = flangedZ(f, hole->c, hole->rho, a);

```

```

/* calculate complex end corrections */
d_flanged = divz(arctanz(divz(Z_flanged, multz(j, Z0))),real(k));
d_cyl = subz(d_flanged, real(0.47 * a * pow(a / b, 0.8)));

/* add empirical correction */
d_cyl = addz(d_cyl, real(CORR_OPEN_FINGER_HOLE_LENGTH * a));

Z = multz(j, multz(Z0, tanz(multz(real(k), d_cyl))));
return Z;
}

complex openKeyedHoleLoadZ(double f, Hole hole) {
    double a, d, q, h, e, w;
    double d_corr, d_e_on_a, k;
    complex Z_circ, Z0, d_circ, d_disk, Z, R;
    Key key = hole->key;

    a = hole->radius;
    d = key->radius;
    q = key->holeRadius;
    h = key->height;
    e = key->thickness;
    w = key->wallThickness;

    /* hack for classical flutes since the keyed holes have no
       chimneys */
    if (key->chimneyHeight == 0) w = DBL_MAX;

    k = (2*M_PI*f)/hole->c;

    /* calculate the characteristic impedance */
    Z0 = charZ(hole->c, hole->rho, a);

    /* Dalmont et al. (2001) Radiation impedance of tubes with different
       flanges: Numerical and experimental investigations, Journal of
       Sound and Vibration 244(3), 505--534, eqs. (48, 51, 52) */
    d_corr = a / (3.5 * pow(h/a, 0.8) * pow(h/a + 3 * w/a, -0.4) + 30
                  * pow(h/d, 2.6));
    /* add empirical length correction */
    d_corr = d_corr + CORR_OPEN_KEYED_HOLE_LENGTH * a;

    if (q > 0) {
        d_e_on_a = 1.64*a/q - 0.15*a/d - 1.1 + e*a/(q*q);
        d_corr = d_corr / (1 + 5*pow(d_e_on_a, -1.35)*pow(h/a, -0.2));
    }

    Z_circ = radiationZ(f, hole->c, hole->rho, a, w/a);

    /* calculate complex end corrections */
    d_circ = divz(arctanz(divz(Z_circ, multz(j, Z0))),real(k));
    d_disk = addz(d_circ, real(d_corr));
    Z = multz(j, multz(Z0, tanz(multz(real(k), d_disk))));

    /* add empirical resistance */
    R = multz(Z0, real(0.4 * pow(k * a, 2)));
    Z = addz(Z, R);
}

```

```

    return Z;
}

complex holeInnerRadiationImpedance(double f, Hole hole) {
    double t_i;
    double Z0 = charZ(hole->c, hole->rho, hole->radius).Re;
    double k = (2*M_PI*f)/hole->c;

    t_i = innerRadiationLengthCorrection(hole->boreRadius,
                                          hole->radius);
    return imaginary(t_i * k * Z0);
}

complex holeSeriesImpedance(double f, Hole hole) {
    double a = hole->boreRadius;
    double b = hole->radius;
    double delta = b / a;
    double t = hole->length, t_0 = 0, t_a;
    double Z0 = charZ(hole->c, hole->rho, hole->boreRadius).Re;
    double k = (2*M_PI*f)/hole->c;

    if (strcmp(hole->fingering, "CLOSED") == 0) {
        if (hole->key == NULL)
            t_0 = b * (0.55 - 0.15 / cosh(9 * t / a) + 0.4
                       / cosh(6.5 * t / a) * (delta - 1));
        t_a = closedHoleSeriesLengthCorrection(a, b, t - t_0);
    }
    else
        t_a = openHoleSeriesLengthCorrection(a, b);

    return imaginary(t_a * k * Z0);
}

TransferMatrix embouchureMatrix(double f, EmbouchureHole h,
                                double entryratio, complex branchZ) {
    TransferMatrix m, riserMatrix, cornerMatrix, innerRadMatrix;
    double t_m, t_i, t_a;
    double radiusin, radiusout;
    complex Z_i, Z_a;
    double Z0_hole = charZ(h->c, h->rho, h->radiusin).Re;
    double Z0_bore = charZ(h->c, h->rho, h->boreRadius).Re;
    double k = (2*M_PI*f)/h->c;

    /* calculate the matching length correction */
    t_m = matchingLengthCorrection(h->boreRadius, h->radiusin);

    /* introduce lossy elements to account for the discontinuity */
    m = identitym();
    m->B = embouchureSeriesResistance(f, h, entryratio);
    m->C = embouchureShuntConductance(f, h, entryratio);

    /* calculate the matrix (with losses) for the tube section
       comprising the hole and matching length */
    radiusin = h->radiusin;
    radiusout = entryratio * h->radiusout;

    riserMatrix = (radiusin == radiusout) ?
        tubeMatrix(f, h->c, h->rho, h->length + t_m, radiusin, 1) :

```

```

coneMatrix(f, h->c, h->rho, h->length + t_m, radiusout, radiusin,
           1);
rmultm(m, riserMatrix);

/* calculate the inner radiation impedance */
t_i = innerRadiationLengthCorrection(h->boreRadius, h->radiusin);
/* add extra length correction for the embouchure hole */
t_i = t_i + embouchureLengthCorrection(h->boreRadius, h->radiusin);
Z_i = imaginary(t_i * k * Z0_hole);

/* add the inner radiation impedance to the matrix m by
   multiplication */
innerRadMatrix = identitym();
innerRadMatrix->B = Z_i;
rmultm(m, innerRadMatrix);

/* calculate the series impedance */
t_a = openHoleSeriesLengthCorrection(h->boreRadius, h->radiusin);
Z_a = imaginary(t_a * k * Z0_bore);

/* add half the series impedance to the branch impedance */
branchZ = addz(branchZ, divz(Z_a, real(2.0)));

/* multiply m by matrix representing the corner */
cornerMatrix = identitym();
cornerMatrix->C = divz(one, branchZ);
cornerMatrix->B = divz(Z_a, real(2.0));
rmultm(m, cornerMatrix);
return m;
}

double embouchureLengthCorrection(double a, double b) {
    double delta = b / a;
    return b * 0.5 * pow(delta, 2);
}

complex embouchureSeriesResistance(double f, EmbouchureHole h,
                                    double entryratio) {
    double Z0 = charZ(h->c, h->rho, entryratio * h->radiusout).Re;
    return real(Z0 * 6.9e-6 * f);
}

complex embouchureShuntConductance(double f, EmbouchureHole h,
                                    double entryratio) {
    double Z0 = charZ(h->c, h->rho, entryratio * h->radiusout).Re;
    return real(1.3e-4 * f / Z0);
}

double matchingLengthCorrection(double a, double b) {
    /* Dalmont et al. (2002) Experimental Determination of the
       Equivalent Circuit of an Open Side Hole: Linear and Non Linear
       Behaviour, eqn. (6) */
    /* Nederveen et al. (1998) Corrections for woodwind tone-holes, eqn.
       (37) */
    double delta = b / a;
    return b * delta * (1 + 0.207 * pow(delta, 3)) / 8;
}

```

```
double innerRadiationLengthCorrection(double a, double b) {
    double delta = b / a;
    /* Dalmont et al. (2002), eqn. (4) */
    /* Nederveen et al. (1998), eqn. (40) */
    return b * (0.82 - 1.4 * pow(delta, 2) + 0.75 * pow(delta, 2.7));
}

double closedHoleSeriesLengthCorrection(double a, double b,
    double t) {
    double delta = b / a;

    if (t > 0)
        return -b * pow(delta, 2) / (1.78 / tanh(1.84 * t / b) + 0.940 +
            0.540 * delta + 0.285 * pow(delta, 2));
    else if (t < 0)
        return -b * pow(delta, 2) / (1.78 / (1.84 * t / b) + 0.940 +
            0.540 * delta + 0.285 * pow(delta, 2));
    else
        return 0;
}

double openHoleSeriesLengthCorrection(double a, double b) {
    double delta = b / a;

    /* Dubos et al. (1999), eqn. (74), modified to remove dependence on
       t */
    return -b * pow(delta, 2) / (1.78 + 0.940 + 0.540 * delta + 0.285
        * pow(delta, 2));
}
```

Listing B.32: Woodwind.dtd

```
<!-- woodwind.dtd -->
<!-- Paul Dickens - February 2007 -->
<!-- Data Type Definition for a woodwind -->

<!ELEMENT woodwind ((embouchurehole, upstream)?, downstream)>
<!ATTLIST woodwind description CDATA #IMPLIED>

<!ELEMENT embouchurehole (radiusin, radiusout, length, boreradius)>
<!ELEMENT upstream (bore+)>
<!ATTLIST upstream flange CDATA #REQUIRED>

<!ELEMENT downstream (bore, (hole | bore)*)>
<!ATTLIST downstream flange CDATA #REQUIRED>

<!ELEMENT bore (radius1, radius2, length)>
<!ELEMENT hole (radius, length, boreradius, key?)>
<!ATTLIST hole name CDATA #IMPLIED>

<!ELEMENT key (radius, holeradius, height, thickness, wallThickness,
chimneyHeight)>

<!ELEMENT radius (#PCDATA)>
<!ELEMENT radiusin (#PCDATA)>
<!ELEMENT radiusout (#PCDATA)>
<!ELEMENT radius1 (#PCDATA)>
<!ELEMENT radius2 (#PCDATA)>
<!ELEMENT boreradius (#PCDATA)>
<!ELEMENT length (#PCDATA)>
<!ELEMENT holeradius (#PCDATA)>
<!ELEMENT height (#PCDATA)>
<!ELEMENT thickness (#PCDATA)>
<!ELEMENT wallThickness (#PCDATA)>
<!ELEMENT chimneyHeight (#PCDATA)>
```

Appendix C

Quantifying music

The musician has words to describe the perceptual qualities of sound—words such as *pitch*, *volume* and *timbre*. The physicist or mathematician has her own set of words to describe sound—such as *amplitude*, *frequency*, *speed*, *wavelength*, *spectral envelope* and *transient*. To an extent, this is perfectly natural, since the musician and the physicist are concerned with different things. However, no two fields of inquiry are ever truly separate, and the union of music, physics and mathematics has been a particularly fruitful one. The premise of this thesis is that we can analyse a flute in order to give useful information to instrument makers, who are thus enabled to make instruments better suited to the particular requirements of musicians. Therefore, a brief discussion of the relationship between music and physics is in order.

The musician's word *pitch* is related to the physicist's word *frequency*, the number of times a periodic wave repeats itself in a fixed time interval. Frequency is measured in hertz (Hz). Low pitched notes have low frequencies, and high pitched notes high frequencies. The note A4 (the A above middle C on a piano) usually has a nominal frequency of 440 Hz, and the frequencies of all other notes are calculated based on this reference value. (Often, the frequency of other notes is calculated according to the equal-tempered scale. Woodwind makers usually design their instruments to play in many different keys, and the equal-tempered scale is a compromise.) An *octave* in music is an interval between two notes that differ in frequency by a factor of two. In the equal-tempered scale a semitone corresponds to a frequency ratio of $2^{1/12}$. The semitone is further divided into 100 *cents* ($2^{1/1200}$).

The *volume* of a musical sound is related to the *amplitude* of the physical wave, but because human perception of sound is frequency-dependent it depends also on the spectral envelope and on some details of the temporal envelope, such as attack transients and vibrato. The frequency range of hearing is from 20 Hz to 20 kHz with a peak at around 1–3 kHz. Filters with weighting factors exist to approximate the frequency response of the human ear (called A, B and C weighting factors). Ears (like microphones) respond to changes in air pressure and so sound amplitude is measured in pascals (Pa). Sound levels are commonly given in decibels (dB), which is a logarithm of a pressure ratio.

Perhaps the most difficult musical term to relate to physics is the *timbre* of a sound. Timbre is defined negatively: it is that quality which differs between two sounds of equal pitch and loudness. Timbre has some relation to the *spectral components* of the wave. A musical note with a frequency of 440 Hz is usually a mixture of *harmonics*, that is, the wave will contain contributions at frequencies of 880 Hz, 1320 Hz etc. The relative strength of these harmonics differ among different instruments, different volumes and among different notes on the same instrument. A sound with many strong harmonics is generally considered *bright* while a sound with a strong fundamental frequency and few harmonics is considered more *mellow* or *soft*. But

spectral content is far from the only determinant of timbre. The initial transients in a musical note are also very important in distinguishing between, say, a note played on a flute and the same note played on a clarinet. In this work we make some predictions about aspects of the timbre of notes based on mathematical modelling of flutes, but it is impossible to convey the richness of a concept such as timbre in a single numerical prediction.

Musical instruments may all be thought of as devices that convert one form of energy to another. Energy is supplied to the instrument in a variety of forms, and some of it is converted into sound waves. As energy converters (or *transducers*), musical instruments are extremely inefficient, typically converting only 1% of input energy into sound (the remainder is lost as heat, or converted into a constant flow of air, which we do not perceive as sound). The input energy of a guitar, for instance, is supplied by the guitarist plucking a string; this is then transferred from the string to the soundboard and from the soundboard to the surrounding air. In woodwind instruments, the input energy is carried by air, either as a flow or pressure. To sound a flute the flautist blows a stream of air across the embouchure and to sound a clarinet, the player supplies a pressure at the embouchure end.

References

- Åbom, M. & Bodén, H. (1988), 'Error analysis of two-microphone measurements in ducts with flow', *J. Acoust. Soc. Am.* **83**(6), 2429–38.
- Ando, Y. (1970), 'On the sound radiation from semi-infinite circular pipe of certain wall thickness', *Acustica* **22**(4), 219–25.
- Backus, J. (1964), 'Effect of wall material on the steady-state tone quality of woodwind instruments', *J. Acoust. Soc. Am.* **36**(10), 1881–7.
- Backus, J. (1974), 'Input impedance curves for the reed woodwind instruments', *J. Acoust. Soc. Am.* **56**(4), 1266–79.
- Backus, J. (1976), 'Input impedance curves for the brass instruments', *J. Acoust. Soc. Am.* **60**(2), 470–80.
- Baines, A. (1967), *Woodwind Instruments and their History*, Faber and Faber, London.
- Bate, P. (1975), *The Flute*, Ernest Benn, London.
- Benade, A. H. (1960), 'On the mathematical theory of woodwind finger holes', *J. Acoust. Soc. Am.* **32**(12), 1591–1608.
- Benade, A. H. (1968), 'On the propagation of sound waves in a cylindrical conduit', *J. Acoust. Soc. Am.* **44**(2), 616–23.
- Benade, A. H. (1976), *Fundamentals of Musical Acoustics*, Oxford University Press.
- Benade, A. H. & French, J. W. (1965), 'Analysis of the flute head joint', *J. Acoust. Soc. Am.* **37**(4), 679–691.
- Benade, A. H. & Ibisi, M. I. (1987), 'Survey of impedance methods and a new piezo-disk-driven impedance head for air columns', *J. Acoust. Soc. Am.* **81**(4), 1152–67.
- Benade, A. H. & Murday, J. S. (1967), 'Measured end corrections for woodwind tone-holes', *J. Acoust. Soc. Am.* **41**(6), 1609.
- Beranek, L. L. (1954), *Acoustics*, McGraw-Hill, New York.
- Bernard, M. & Denardo, B. (1996), 'Re-computation of Ando's approximation of the end correction for a radiating semi-infinite circular pipe', *Acustica – acta acustica* **82**(4), 670–1.
- Bodén, H. & Åbom, M. (1986), 'Influence of errors on the two-microphone method for measuring acoustic properties in ducts', *J. Acoust. Soc. Am.* **79**(2), 541–9.
- Boehm, T. (1871), *The Flute and Flute-playing*, translated by D.C. Miller, Dover, New York.
- Botros, A. M. (2001), Data mining for alternate fingerings and multiphonics of the modern flute, Bachelor of Engineering (Computer) thesis, School of Computer Science and Engineering, University of New South Wales.
- Botros, A., Smith, J. & Wolfe, J. (2002), 'The Virtual Boehm Flute—a web service that predicts multiphonics, microtones and alternative fingerings', *Acoustics Australia* **30**(2), 61–5.

- Botros, A., Smith, J. & Wolfe, J. (2006), 'The Virtual Flute: an advanced fingering guide generated via machine intelligence', *Journal of New Music Research* **35**(3), 183–96.
- Brass, D. & Locke, A. (1997), 'The effect of the evanescent wave upon acoustic measurements in the human ear canal', *J. Acoust. Soc. Am.* **101**(4), 2164–75.
- Bruneau, A. M. (1987), 'An acoustic impedance sensor with two reciprocal transducers', *J. Acoust. Soc. Am.* **81**(4), 1168–78.
- Caussé, R., Kergomard, J. & Lurton, X. (1984), 'Input impedance of brass musical instruments—comparison between experiment and numerical models', *J. Acoust. Soc. Am.* **75**(1), 241–54.
- Chung, J. Y. & Blaser, D. A. (1980a), 'Transfer function method of measuring acoustic intensity in a duct system with flow', *J. Acoust. Soc. Am.* **68**(6), 1570–7.
- Chung, J. Y. & Blaser, D. A. (1980b), 'Transfer function method of measuring in-duct acoustic properties. I. Theory', *J. Acoust. Soc. Am.* **68**(3), 907–13.
- Chung, J. Y. & Blaser, D. A. (1980c), 'Transfer function method of measuring in-duct acoustic properties. II. Experiment', *J. Acoust. Soc. Am.* **68**(3), 914–21.
- Coltman, J. W. (1966), 'Resonance and sounding frequencies of the flute', *J. Acoust. Soc. Am.* **40**(1), 98–107.
- Coltman, J. W. (1968a), 'Acoustics of the flute', *Physics Today* **21**(11), 25–33.
- Coltman, J. W. (1968b), 'Sounding mechanism of the flute and organ pipe', *J. Acoust. Soc. Am.* **44**(4), 983–92.
- Coltman, J. W. (1971), 'Effect of material on flute tone quality', *J. Acoust. Soc. Am.* **49**(2), 520–3.
- Coltman, J. W. (1979), 'Acoustical analysis of the Boehm flute', *J. Acoust. Soc. Am.* **65**(2), 499–506.
- Coltman, J. W. (1998), 'The John W. Coltman archive', <<http://ccrma.stanford.edu/marl/Coltman/computer.html>>. Viewed 10 March 2007.
- Cramer, O. (1993), 'The variation of the specific heat ratio and the speed of sound in air with temperature, pressure, humidity, and CO₂ concentration', *J. Acoust. Soc. Am.* **93**(5), 2510–16.
- Dalmont, J.-P. (2001a), 'Acoustic impedance measurement, Part I: A review', *J. Sound Vib.* **243**(3), 427–39.
- Dalmont, J.-P. (2001b), 'Acoustic impedance measurement, Part II: A new calibration method', *J. Sound Vib.* **243**(3), 441–59.
- Dalmont, J.-P., Nederveen, C. J., Dubos, V., Ollivier, S., Meserette, V. & te Sligte, E. (2002), 'Experimental determination of the equivalent circuit of an open side hole: linear and nonlinear behaviour', *Acustica – acta acustica* **88**(4), 567–75.
- Dalmont, J.-P., Nederveen, C. J. & Joly, N. (2001), 'Radiation impedance of tubes with different flanges: Numerical and experimental investigations', *J. Sound Vib.* **244**(3), 505–34.
- Dauvois, M., Boutillon, X., Fabre, B. & Verge, M.-P. (1998), 'Son et musique au paléolithique', *Pour La Science* **253**, 52–58.
- Dick, R. (1999), 'Why I love the Cooper scale', <<http://www.larrykrantz.com/rdick2.htm>>. Viewed 2 June 2007.
- Donne, J. (1971), *The Complete English Poems*, ed. A. J. Smith, Penguin Books.

- Dubos, V., Kergomard, J., Khettabi, A., Dalmont, J.-P., Keefe, D. H. & Nederveen, C. J. (1999), 'Theory of sound propagation in a duct with a branched tube using modal decomposition', *Acustica – acta acustica* **85**(2), 153–69.
- Elliott, S., Bowsher, J. & Watkinson, P. (1982), 'Input and transfer response of brass wind instruments', *J. Acoust. Soc. Am.* **72**(6), 1747–60.
- Fabre, B. & Hirschberg, A. (2000), 'Physical modeling of flue instruments: A review of lumped models', *Acta Acustica* **86**(4), 599–610.
- Fletcher, N. H. (1975), 'Acoustical correlates of flute performance technique', *J. Acoust. Soc. Am.* **57**(1), 233–7.
- Fletcher, N. H. (1976), 'Jet-drive mechanism in organ pipes', *J. Acoust. Soc. Am.* **60**(2), 481–3.
- Fletcher, N. H. & Rossing, T. D. (1998), *The Physics of Musical Instruments*, 2nd edn, Springer-Verlag New York, Inc.
- Fletcher, N. H., Smith, J., Tarnopolsky, A. Z. & Wolfe, J. (2005), 'Acoustic impedance measurements—correction for probe geometry mismatch', *J. Acoust. Soc. Am.* **117**(5), 2889–95.
- Fletcher, N. H., Strong, W. J. & Silk, R. K. (1982), 'Acoustical characterization of flute head joints', *J. Acoust. Soc. Am.* **71**(5), 1255–60.
- Fletcher, N. H. & Thwaites, S. (1979), 'Wave propagation on an acoustically perturbed jet', *Acustica* **42**(5), 323–34.
- Fletcher, N. H. & Thwaites, S. (1983), 'The physics of organ pipes', *Scientific American* **248**(1), 84–93.
- Fredberg, J. J., Wohl, M. E. B., Glass, G. M. & Dorkin, H. L. (1980), 'Airway area by acoustic reflections measured at the mouth', *J. Appl. Physiol.* **48**(5), 749–58.
- Fritz, C. & Wolfe, J. (2005), 'How do clarinet players adjust the resonances of their vocal tracts for different playing effects?', *J. Acoust. Soc. Am.* **118**(5), 3306–15.
- Giacomo, P. (1982), 'Equation for the determination of the density of moist air', *Metrologia* **18**(1), 33–40.
- Gibiat, V. & Laloë, F. (1990), 'Acoustical impedance measurements by the two-microphone-three-calibration (TMTC) method', *J. Acoust. Soc. Am.* **88**(6), 2533–45.
- Jang, S.-H. & Ih, J.-G. (1998), 'On the multiple microphone method for measuring in-duct acoustic properties in the presence of mean flow', *J. Acoust. Soc. Am.* **103**(3), 1520–6.
- Keefe, D. H. (1982a), 'Experiments on the single woodwind tone hole', *J. Acoust. Soc. Am.* **72**(3), 688–99.
- Keefe, D. H. (1982b), 'Theory of the single woodwind tone hole', *J. Acoust. Soc. Am.* **72**(3), 676–87.
- Keefe, D. H. (1984), 'Acoustical wave propagation in cylindrical ducts: transmission line parameter approximations for isothermal and nonisothermal boundary conditions', *J. Acoust. Soc. Am.* **75**(1), 58–62.
- Keefe, D. H. (1990), 'Woodwind air column models', *J. Acoust. Soc. Am.* **88**(1), 35–51.

- Kob, M. & Neuschaefer-Rube, C. (2002), 'A method for measurement of the vocal tract impedance at the mouth', *Med. Eng. Phys.* **24**(7–8), 467–71.
- Koestler, A. (1959), *The Sleepwalkers*, Penguin, London.
- Levine, H. & Schwinger, J. (1948), 'On the radiation of sound from an unflanged circular pipe', *Physical Review* **73**, 383–406.
- Nederveen, C. J. (1998), *Acoustical aspects of woodwind instruments*, 2nd edn, Northern Illinois University Press.
- Nederveen, C. J., Jansen, J. K. M. & van Hassel, R. R. (1998), 'Corrections for woodwind tone-hole calculations', *Acustica – acta acustica* **84**(5), 957–66.
- Nomura, Y., Yamamura, I. & Inawashiro, S. (1960), 'On the acoustic radiation from a flanged circular pipe', *Journal of the Physical Society of Japan* **15**(3), 510–17.
- Norris, A. N. & Sheng, I. C. (1989), 'Acoustic radiation from a circular pipe with an infinite flange', *J. Sound Vib.* **135**(1), 85–93.
- Pagneux, V., Amir, N. & Kergomard, J. (1996), 'A study of wave propagation in varying cross-section waveguides by modal decomposition. Part I. Theory and validation', *J. Acoust. Soc. Am.* **100**(4), 2034–48.
- Plitnik, G. R. & Strong, W. J. (1979), 'Numerical method for calculating input impedances of the oboe', *J. Acoust. Soc. Am.* **65**(3), 816–25.
- Porter, W. M. (2005), 'Fingering chart for McGee "Rudall perfected" six-key flute', available on Terry McGee's website <http://www.mcgee-flutes.com/Modern_6-key_Fingering_Chart.html>. Viewed 10 June 2007. A copy of this website archived by the National Library of Australia is available at <<http://nla.gov.au/nla.arc-24785>>.
- Poulton, G. (2005), An analysis of undercut toneholes in woodwinds, in 'Proceedings of the Australian Institute of Physics 16th Biennial Congress', Canberra, Australia.
- Pratt, R. L., Elliott, S. J. & Bowsher, J. M. (1977), 'The measurement of the acoustic impedance of brass instruments', *Acustica* **38**(4), 236–46.
- Rayleigh, L. (1894), *The Theory of Sound*, Macmillan, London.
- Rockstro, R. S. (1890), *A Treatise on the Flute*, Reprinted by Musica Rara, London, 1967.
- Seybert, A. F. & Ross, D. F. (1977), 'Experimental determination of acoustic properties using a two-microphone random-excitation technique', *J. Acoust. Soc. Am.* **61**(5), 1362–70.
- Singh, R. & Schary, M. (1978), 'Acoustic impedance measurement using sine sweep excitation and known volume velocity technique', *J. Acoust. Soc. Am.* **64**(4), 995–1003.
- Smith, J. R. (1995), 'Phasing of harmonic components to optimize measured signal-to-noise ratios of transfer functions', *Meas. Sci. Technol.* **6**(9), 1343–8.
- Smith, J. R., Henrich, N. & Wolfe, J. (1997), 'The acoustic impedance of the Boehm flute: Standard and some non-standard fingerings', *Proc. Inst. Acoustics* **19**, 315–30.
- Smith, J., Rey, G., Dickens, P., Fletcher, N., Hollenberg, L. & Wolfe, J. (2007), 'Vocal tract resonances and the sound of the Australian didjeridu (yidaki). III. Determinants of playing quality', *J. Acoust. Soc. Am.* **121**, 547–58.

- Strong, W. J., Fletcher, N. H. & Silk, R. K. (1985), 'Numerical calculation of flute impedances and standing waves', *J. Acoust. Soc. Am.* **77**(6), 2166–72.
- Thwaites, S. & Fletcher, N. H. (1980), 'Wave propagation on turbulent jets', *Acustica* **45**(3), 175–9.
- Thwaites, S. & Fletcher, N. H. (1982), 'Wave propagation on turbulent jets. II. Growth', *Acustica* **51**(1), 44–9.
- Thwaites, S. & Fletcher, N. H. (1983), 'Acoustic admittance of organ pipe jets', *J. Acoust. Soc. Am.* **74**(2), 400–8.
- van der Eerden, F. J. M., de Bree, H.-E. & Tijdeman, H. (1998), 'Experiments with a new acoustic particle velocity sensor in an impedance tube', *Sensors and Actuators A (Physical)* **A69**(2), 126–33.
- van Walstijn, M., Campbell, M., Kemp, J. & Sharp, D. (2005), 'Wideband measurement of the acoustic impedance of tubular objects', *Acustica – acta acustica* **91**(3), 590–604.
- Vergez, C., de la Cuadra, P. & Fabre, B. (2005), Jet motion in flute-like instruments: experimental investigation through flow visualization and image processing, in 'Forum Acusticum', S. Hirzel, Budapest, Hungary, p. S45.
- Ward, W. D. (1954), 'Subjective musical pitch', *J. Acoust. Soc. Am.* **26**, 369–80.
- Watson, A. P. & Bowsher, J. M. (1988), 'Impulse measurements on brass musical instruments', *Acustica* **66**(3), 170–4.
- Widholm, G., Linortner, R., Kausel, W. & Bertsch, M. (2001), Silver, gold, platinum—and the sound of the flute, in 'Proceedings of ISMA (International Symposium on Musical Acoustics) 2001', pp. S.277–80.
- Wolfe, J. & Smith, J. (2003), 'Cutoff frequencies and cross fingerings in baroque, classical, and modern flutes', *J. Acoust. Soc. Am.* **114**(4), 2263–72.
- Wolfe, J., Smith, J. & Green, M. (2003), The effects of placement of the head joint stopper on the impedance spectra of transverse flutes, in 'Proc. Eighth Western Pacific Acoustics Conference, Melbourne. (C. Don, ed.) Aust. Acoust. Soc., Castlemaine, Aust.'
- Wolfe, J., Smith, J., Tann, J. & Fletcher, N. H. (2001a), 'Acoustic impedance spectra of classical and modern flutes', *J. Sound Vib.* **243**(1), 127–44.
- Wolfe, J., Smith, J., Tann, J. & Fletcher, N. H. (2001b), 'Acoustics of classical and modern flutes: a compendium of impedance spectra, sound spectra, sounds and fingerings', JSV+ (electronic supplement at <http://journals.harcourt-international.com/journals/jsv/supplementary/suppindex.htm>).