Use of Pattern Classification Algorithms to Interpret Passive and Active Data Streams from a Walking-Speed Robotic Sensor Platform

Eric Allen Dieckman

College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Acoustics, Dynamics, and Controls Commons, and the Robotics Commons

Recommended Citation


https://dx.doi.org/doi:10.21220/s2-5cy6-jk36

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.
Use of Pattern Classification Algorithms to Interpret Passive and Active Data Streams from a Walking-Speed Robotic Sensor Platform

Eric Allen Dieckman
Wellington, Missouri

M.S. Architectural Acoustics, Rensselaer Polytechnic Institute, 2009
B.S. Physics, Truman State University, 2008

A Dissertation presented to the Graduate Faculty of the College of William and Mary in Candidacy for the Degree of Doctor of Philosophy

Department of Applied Science

The College of William and Mary
May 2014
This Dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Approved by the Committee, October 2013

Committee Chair
Professor Mark Hinders, Applied Science
The College of William and Mary

Professor Gunter Luepke, Applied Science
The College of William and Mary

Research Assistant Professor Saskia Mordjick, Computer Science
The College of William and Mary

Dr. William Fehlman II
U.S. Army Capabilities Integration Center
ABSTRACT

In order to perform useful tasks for us, robots must have the ability to notice, recognize, and respond to objects and events in their environment. This requires the acquisition and synthesis of information from a variety of sensors. Here we investigate the performance of a number of sensor modalities in an unstructured outdoor environment, including the Microsoft Kinect, thermal infrared camera, and coffee can radar. Special attention is given to acoustic echolocation measurements of approaching vehicles, where an acoustic parametric array propagates an audible signal to the oncoming target and the Kinect microphone array records the reflected backscattered signal. Although useful information about the target is hidden inside the noisy time domain measurements, the Dynamic Wavelet Fingerprint process (DWFP) is used to create a time-frequency representation of the data. A small-dimensional feature vector is created for each measurement using an intelligent feature selection process for use in statistical pattern classification routines. Using out experimentally measured data from real vehicles at 50 m, this process is able to correctly classify vehicles into one of five classes with 94% accuracy. Fully three-dimensional simulations allow us to study the nonlinear beam propagation and interaction with real-world targets to improve classification results.
# Table of Contents

Acknowledgments iv  
Dedication v  
List of Figures vi  
List of Tables x  
List of Symbols xii  

1 Overview 1  

2 Finding information in noisy data 6  
2.1 Time-frequency analysis of signals 7  
2.1.1 Fourier transform 7  
2.1.2 Short-time Fourier transform 8  
2.1.3 Other methods of time-frequency analysis 10  
2.1.4 Wavelets 11  
2.2 The Dynamic Wavelet Fingerprint (DWFP) 18  
2.2.1 Feature creation 19  
2.2.2 Feature extraction 22  
2.3 An application: Ultrasonic detection of flaws in microelectronics 24  
2.3.1 Background 25  
2.3.2 Ultrasonic measurements of microelectronics 25  
2.3.3 Creating flawed microelectronics samples 30  
2.3.4 DWFP analysis 33  
2.3.5 Application to analysis of other signals 38  

3 Statistical pattern classification 40  
3.1 Statistical pattern classification by example: analysis of glass 41  
3.2 Feature extraction and selection 47  
3.3 Classification 50  
3.3.1 Parametric classifiers 51  
3.3.2 Nonparametric classifiers 53  
3.3.3 Improving classifier performance 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Visualizing classification results</td>
<td>64</td>
</tr>
<tr>
<td>3.5</td>
<td>Some results</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>rMary - a walking-speed mobile sensor platform</td>
<td>72</td>
</tr>
<tr>
<td>4.1</td>
<td>Towards an autonomous walking-speed robotic platform</td>
<td>74</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Sensor modalities for mobile robots</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>Investigation of sensor modalities using rMary</td>
<td>77</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Thermal infrared (IR)</td>
<td>79</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Kinect</td>
<td>82</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Audio</td>
<td>93</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Radar</td>
<td>98</td>
</tr>
<tr>
<td>5</td>
<td>Acoustic echolocation from a mobile robot</td>
<td>102</td>
</tr>
<tr>
<td>5.1</td>
<td>Vehicle classification</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td>Acquiring acoustic echolocation data</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Initial data analysis</td>
<td>110</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Detecting a signal reflected from a wall</td>
<td>110</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Data from oblique angles</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>Classification of oncoming vehicles using acoustic echolocation</td>
<td>118</td>
</tr>
<tr>
<td>6.1</td>
<td>Pattern classification</td>
<td>118</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Compiling data</td>
<td>119</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Aligning reflected signals</td>
<td>120</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Feature creation with DWFP</td>
<td>127</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Intelligent feature selection</td>
<td>130</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Statistical pattern classification</td>
<td>135</td>
</tr>
<tr>
<td>6.2</td>
<td>Results</td>
<td>136</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Proof-of-concept: Acoustic classification of stationary vehicles</td>
<td>137</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Acoustic classification of oncoming vehicles</td>
<td>139</td>
</tr>
<tr>
<td>7</td>
<td>Simulations of scattering from a nonlinear acoustic beam</td>
<td>155</td>
</tr>
<tr>
<td>7.1</td>
<td>The acoustic parametric array</td>
<td>156</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Modeling nonlinear acoustic wave propagation</td>
<td>160</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Numerical solutions of the KZK equation</td>
<td>163</td>
</tr>
<tr>
<td>7.2</td>
<td>Simulations of acoustic scattering</td>
<td>170</td>
</tr>
<tr>
<td>7.2.1</td>
<td>AFIT</td>
<td>172</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Implementation of AFIT on the SciClone computing cluster</td>
<td>177</td>
</tr>
<tr>
<td>7.3</td>
<td>Simulations of acoustic scattering from vehicles</td>
<td>179</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Creating vehicle scatterers</td>
<td>180</td>
</tr>
<tr>
<td>7.3.2</td>
<td>The effect of incident pulse length on backscattered reflection</td>
<td>188</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Simulated microphone data from the backscattered pressure field</td>
<td>203</td>
</tr>
<tr>
<td>8</td>
<td>Future directions in sensor fusion for mobile robotics</td>
<td>218</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>8.1 Sensor modalities</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>8.2 Future directions in sensor fusion</td>
<td>220</td>
<td></td>
</tr>
</tbody>
</table>

Appendices  
Bibliography  
Vita
Acknowledgments

I would like to thank my advisor Dr. Mark Hinders for his guidance, assistance, and discussion of any topic from scattering theory to current television shows written for a teenage audience. Jonathan Stevens provided invaluable engineering expertise and Chris Houck collected much of the contact ultrasound data from microelectronics. Members of the NDE lab, especially Lora Weiss, spent many hours standing at the side of the street attempting to collect data with rMary while fielding questions from curious tourists. Expert insight into FDTD simulations was provided by Dr. Jonathan Botts, especially concerning impulse response methods.

I would not have ended up on this path without the direction of my parents, Kevin and Susie Dieckman. My wonderful wife Liz provided encouragement as well as expert carpentry and grommeting skills.

This work was performed in part using computational facilities at the College of William and Mary which were provided with the assistance of the National Science Foundation, the Virginia Port Authority, and Virginia's Commonwealth Technology Research Fund.

Partial funding was provided by a grant from the Bill & Melinda Gates Foundation through the Grand Challenges Explorations initiative.
This dissertation is dedicated to my wife, Liz, the real doctor of the family.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A spectrogram image provides both time and frequency information</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>The Gibbs phenomenon shows the difficulties of using continuous functions to approximate sharp discontinuities</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Wavelet decomposition can be used to filter a signal</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of time and frequency resolutions of the STFT and wavelet transform</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>The dynamic wavelet fingerprint process (DWFP) maps a one-dimensional signal to a two-dimensional ternary image</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Labeling of individual fingerprint</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Ultrasound A-, B-, and C- scans of a microelectronic component</td>
<td>27</td>
</tr>
<tr>
<td>2.8</td>
<td>Delay lines reduce the footprint of a transducer</td>
<td>28</td>
</tr>
<tr>
<td>2.9</td>
<td>A stabilizer arm holds the transducer assembly for measurements</td>
<td>29</td>
</tr>
<tr>
<td>2.10</td>
<td>Ultrasonic measurement setup</td>
<td>30</td>
</tr>
<tr>
<td>2.11</td>
<td>Completed ultrasonic test platform for microelectronic components</td>
<td>31</td>
</tr>
<tr>
<td>2.12</td>
<td>A Matlab GUI assists in DWFP analysis of ultrasonic measurements</td>
<td>34</td>
</tr>
<tr>
<td>2.13</td>
<td>Flaws in microelectronics are visible in the fingerprint image</td>
<td>36</td>
</tr>
<tr>
<td>2.14</td>
<td>Measurements across an entire component are automatically analyzed to make a flaw/no flaw decision</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Flowchart of the pattern classification process</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Low-dimensional decision boundaries often overlap for high-dimensional data</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Increasing the dimension in which decision boundaries are drawn can more clearly separate data high-dimensional</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Decision boundaries for parametric classifiers are shown for a subset of the glass dataset</td>
<td>54</td>
</tr>
<tr>
<td>3.5</td>
<td>Decision boundaries for parametric classifiers are shown for a subset of the glass dataset with reduced classes</td>
<td>55</td>
</tr>
<tr>
<td>3.6</td>
<td>Decision boundaries for nonparametric classifiers are shown for a subset of the glass dataset</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Decision boundaries for nonparametric classifiers are shown for a subset of the glass dataset with reduced classes</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>Convergence of classification performance for multiple runs</td>
<td>63</td>
</tr>
<tr>
<td>3.9</td>
<td>Example of an average confusion matrix for the glass dataset</td>
<td>66</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.10</td>
<td>ROC curve for binary classification of a subset of the glass dataset</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>rMary robotic sensor platform</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>Sensors on rMary</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>Blackbody radiation curves</td>
<td>81</td>
</tr>
<tr>
<td>4.4</td>
<td>Thermal infrared images of passive objects</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Microsoft Kinect sensor</td>
<td>84</td>
</tr>
<tr>
<td>4.6</td>
<td>Kinect depth-mapping indoors</td>
<td>86</td>
</tr>
<tr>
<td>4.7</td>
<td>Kinect depth-mapping point cloud images</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>Kinect infrared and depth-mapped images outdoors</td>
<td>88</td>
</tr>
<tr>
<td>4.9</td>
<td>Infrared spectrum</td>
<td>89</td>
</tr>
<tr>
<td>4.10</td>
<td>Solar radiation spectrum</td>
<td>90</td>
</tr>
<tr>
<td>4.11</td>
<td>Effect of sunlight on Kinect depth mapping</td>
<td>91</td>
</tr>
<tr>
<td>4.12</td>
<td>Kinect depth mapping outdoors at night</td>
<td>92</td>
</tr>
<tr>
<td>4.13</td>
<td>rWilliam sensor platform</td>
<td>93</td>
</tr>
<tr>
<td>4.14</td>
<td>Nonlinear propagation of sound from an acoustic parametric array</td>
<td>95</td>
</tr>
<tr>
<td>4.15</td>
<td>Acoustic parametric array directionality</td>
<td>97</td>
</tr>
<tr>
<td>4.16</td>
<td>Comparison of the Kinect microphone array and parabolic dish microphone</td>
<td>98</td>
</tr>
<tr>
<td>4.17</td>
<td>Coffee can radar</td>
<td>99</td>
</tr>
<tr>
<td>4.18</td>
<td>Radar ranging of oncoming vehicles</td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>rMary acoustic echolocation</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Labeling vehicles in the recorded microphone data</td>
<td>106</td>
</tr>
<tr>
<td>5.3</td>
<td>Acoustic echolocation data collection locations</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>Reflection of a 5 ms signal incident on a flat wall</td>
<td>113</td>
</tr>
<tr>
<td>5.5</td>
<td>Reflection of a 10 ms signal incident on a flat wall</td>
<td>114</td>
</tr>
<tr>
<td>5.6</td>
<td>Reflection of a 250 ms signal incident on a flat wall</td>
<td>115</td>
</tr>
<tr>
<td>5.7</td>
<td>Oblique reflections from a stationary vehicle</td>
<td>116</td>
</tr>
<tr>
<td>6.1</td>
<td>All four channels of the Kinect microphone array record the same information</td>
<td>120</td>
</tr>
<tr>
<td>6.2</td>
<td>Creation of the signal envelope</td>
<td>123</td>
</tr>
<tr>
<td>6.3</td>
<td>Detecting reflections in a recorded signal using the signal envelope</td>
<td>124</td>
</tr>
<tr>
<td>6.4</td>
<td>Detecting reflections in a recorded signal with cross correlation methods</td>
<td>126</td>
</tr>
<tr>
<td>6.5</td>
<td>A comparison of methods to detect reflections in noisy recorded data</td>
<td>127</td>
</tr>
<tr>
<td>6.6</td>
<td>Fingerprint image creation using the DWFP process</td>
<td>128</td>
</tr>
<tr>
<td>6.7</td>
<td>Diagram of DWFP feature extraction process</td>
<td>129</td>
</tr>
<tr>
<td>6.8</td>
<td>Diagram of feature selection process</td>
<td>132</td>
</tr>
<tr>
<td>6.9</td>
<td>List of selected features</td>
<td>133</td>
</tr>
<tr>
<td>6.10</td>
<td>Diagram of entire feature vector creation process</td>
<td>134</td>
</tr>
<tr>
<td>6.11</td>
<td>Exemplary confusion matrix</td>
<td>136</td>
</tr>
<tr>
<td>6.12</td>
<td>Classification results - 250 ms signal incident on stationary vehicles</td>
<td>138</td>
</tr>
</tbody>
</table>
7.23 Pseudocolor slices of the scattered pressure field through the (x,z) plane for scattering from a bus model ................................................................. 202
7.24 Pseudocolor slices of the scattered pressure field through the (x,z) plane for all five vehicle models - early scattering behavior .................................. 204
7.25 Pseudocolor slices of the scattered pressure field through the (x,z) plane for all five vehicle models - late scattering behavior ............................... 205
7.26 Simulated pressure data recorded near the source plane and 1.6 m into the simulation space ................................................................. 207
7.27 An impulse response method allows easy computation of scattering behavior for any incident signal ................................................................. 208
7.28 A spectrogram image shows that the backscattered reflection retains the structure of a linear chirp ................................................................. 210
7.29 The simulated pressure data looks similar in the time-domain for all vehicle models ................................................................. 212
7.30 Fingerprint images created from simulated pressure signals ................... 213
7.31 Feature selection process ................................................................. 215

8.1 Next generation robotic sensor package ..................................................... 221
LIST OF TABLES

2.1 Wavelet families ............................................................. 16
2.2 Features extracted from the fingerprint image ...................... 23
2.3 List of user parameters in the DWFP feature extraction .......... 24

3.1 Class distribution for the glass dataset ............................... 43
3.2 Each sample in the glass dataset is described by nine values .... 44
3.3 List of classifiers used ..................................................... 61
3.4 Classification results for the glass dataset for varying train/test ratios 62
3.5 Metrics derived from the confusion matrix ......................... 64
3.6 Comparison of overall classification performance for the glass dataset 70
3.7 Overall classification performance for the glass dataset .......... 71

5.1 Acoustic echolocation data collection locations ..................... 108
5.2 Amount of acoustic echolocation data collected from oncoming vehicles 110

6.1 Maximum amount of data in each classification group ............... 121
6.2 Maximum amount of data in each classification group (binned) ... 121
6.3 Wavelets used in the DWFP feature creation process ............... 130
6.4 User parameters in feature selection process ....................... 133
6.5 Effect of prefiltering wavelet choice on classification accuracy .... 142
6.6 Effect of transform wavelet choice on classification accuracy .... 143
6.7 Datasets used for classification tests .................................. 144
6.8 Effect of train/test ratio on classification accuracy ................. 145
6.9 Repeatability of classification results ................................ 147
6.10 Effect of incident signal spacing on classification accuracy ...... 149
6.11 Effect of orientation on classification accuracy ..................... 151
6.12 Comparison of classification accuracy for all incident signals .... 152

7.1 The maximum frequency determines the minimum step sizes for an AFIT simulation ............................................ 178
7.2 Memory requirements for the AFIT simulation depend on the size of the simulation space ..................................... 179
7.3 Parameters used to create vehicle models for the scattering simulation 186
7.4 Simulation space sizes for vehicle models ............................ 189
7.5 Long duration signals require very large AFIT simulation spaces ... 190
7.6 Feature vector components for simulated pressure data
**LIST OF SYMBOLS**

**Chapter 2**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t, \tau$</td>
<td>Time</td>
</tr>
<tr>
<td>$\xi$</td>
<td>Natural frequency</td>
</tr>
<tr>
<td>$\omega$</td>
<td>Angular frequency ($\omega = 2\pi\xi$)</td>
</tr>
<tr>
<td>$\bar{w}(t - \tau)$</td>
<td>Window function</td>
</tr>
<tr>
<td>$\mathcal{F}(\omega)$</td>
<td>Fourier transform</td>
</tr>
<tr>
<td><strong>STFT</strong> ${f(t)}$ $(\tau, \omega)$</td>
<td>Short-time Fourier Transform</td>
</tr>
<tr>
<td>$W(\tau, \omega)$</td>
<td>Wigner-Ville transform</td>
</tr>
<tr>
<td>$G(\tau, \omega)$</td>
<td>Gabor transform</td>
</tr>
<tr>
<td>$D(\tau, \omega)$</td>
<td>Gabor-Wigner transform</td>
</tr>
<tr>
<td>$s$</td>
<td>Wavelet scale</td>
</tr>
<tr>
<td>$\psi(\tau, s)$</td>
<td>Wavelet functions</td>
</tr>
<tr>
<td>$\mathcal{W}(\tau, s)$</td>
<td>Wavelet transform</td>
</tr>
<tr>
<td>$c_{jk}$</td>
<td>Wavelet coefficients</td>
</tr>
</tbody>
</table>

**Chapter 3**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>Number of observations (instances)</td>
</tr>
<tr>
<td>$x_i$</td>
<td>Feature vector, index $i = 1, 2, \ldots, s$</td>
</tr>
<tr>
<td>$\omega_k$</td>
<td>Class $k$</td>
</tr>
<tr>
<td>$\mu_k$</td>
<td>Mean feature vector for class $k$</td>
</tr>
<tr>
<td>$\sigma_k$</td>
<td>Standard deviation of mean feature vector for class $k$</td>
</tr>
<tr>
<td>$S$</td>
<td>Class separation</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>Covariance</td>
</tr>
<tr>
<td>$P(\omega_k</td>
<td>x)$</td>
</tr>
<tr>
<td>$P(\omega_k)$</td>
<td>Prior probability</td>
</tr>
<tr>
<td>$p(x</td>
<td>\omega_k)$</td>
</tr>
<tr>
<td>$g_k(x)$</td>
<td>Discriminant function</td>
</tr>
<tr>
<td>$\hat{p}(x)$</td>
<td>Estimate of class density</td>
</tr>
</tbody>
</table>
Windowing function for Parzen density estimate
Weights for SVM
Sensitivity
Specificity

Chapter 7

Pressure
Density
Particle velocity
Speed of sound
Sound diffusivity
Shear and bulk viscosity
Thermal conductivity
Specific heat at constant volume and pressure
Coefficient of nonlinearity
Retarded time

List of classifiers

Parametric classifiers

qdc Quadratic classifier assuming normal densities
udc Quadratic classifier assuming normal uncorrelated densities
ldc Linear classifier assuming normal densities
klldc Linear classifier by KL expansion of common cov matrix
pcldc Linear classifier by PCA expansion on the joint data
nmc Nearest mean classifier
nmsc Scaled nearest mean classifier
loglc Logistic linear classifier
fisherc Minimum least square linear classifier

Nonparametric classifiers

knnc k-nearest neighbor classifier
parzenc Parzen classifier
parzendc Parzen density based classifier
kernelc General kernel/dissimilarity based classification
perlc Linear perceptron
svc Support vector classifier
nusvc Support vector classifier
rbsvcc Radial basis SV classifier
treeec  Construct binary decision tree classifier
Chapter 1

Overview

In order to perform useful tasks for us, robots must have the ability to notice, recognize, and respond to objects and events in their environment. This is especially true when robots need to operate in unstructured environments. In these situations, robots must sense their environment and make decisions based on this information.

As an example we consider the task of having a robotic assistant fetch a cup of coffee from across the street. While this may be a simple task for a human, an autonomous robotic assistant must exhibit many human behaviors to successfully complete the task. These behaviors include sensing (where is the door?), navigation (how do I get there?), mechanical (how do I open the door?), and human interaction (what do I do if I meet someone else on the way?). In particular, we will focus on the sensing aspects of creating an autonomous robot.

A robot must be able to sense and react to objects and events occurring over a range of distances. For our case of a mobile walking-speed robot, this includes long range sensors that can detect dangers, such as oncoming motor vehicles, in time to evade, as well as close-range sensors that provide more information about stationary objects in the environment. In addition, sensors must be able to provide useful
information in a variety of environmental conditions. While an RGB webcam may provide detailed information in a well lit environment, it is less useful on a foggy night. The key to creating a useful autonomous robot is to equip the robot with a number of complementary sensors so that it can learn about its environment and make decisions.

In particular, we are interested in the use of acoustic echolocation as a long-range sensor modality for mobile robotics. While sonar has long been used as a sensor in underwater environments, the short propagation of ultrasonic waves in air has restricted its use elsewhere. Lower frequency acoustic signals in the audible range are able to propagate long distances in air, but traditional methods of creating highly-directional audible acoustic signals require very large speaker arrays not feasible for a mobile robot. In addition, the complex interactions of these signals with objects in the environment and ubiquitous environmental noise makes the reflected signals very difficult to analyze.

We use an acoustic parametric array to generate our acoustic echolocation signal. This is a physically-small speaker that uses nonlinear acoustics to create a tight beam of low-frequency sound that can propagate long distances. Such a highly directional signal provides good spatial resolution that allows a distinction between the target and environmental clutter. Systematic experimental investigations and simulations allow us to study the propagation of these nonlinear sound beams and their interaction with scatterers.

These sensor signals are very noisy, making it difficult for the robot to extract any useful information. One common technique that can provide additional insight is to transform the problem into an alternate domain. For the simple case of one-dimensional time domain signal this most commonly takes the form of Fourier transform. While this converts the signal to the frequency domain and can reveal previously hidden information, all time domain information is lost in the transformation. A bet-
ter solution for time domain data is to transform the original signal into a joint
time-frequency domain. This can be accomplished by a number of methods, but there
is no one best time-frequency representation. Uncertainty limits restrict simultaneous
time and frequency resolution, some methods are very complex and hard to implement,
and the resulting two-dimensional images can be even more difficult to analyze than
the original signal.

In Chapter 2 we discuss the benefits and problems inherent in various time-
frequency analysis methods and introduce our solution - the Dynamic Wavelet Finger-
print (DWFP). This process uses wavelets to transform a one-dimensional signal into
a two-dimensional binary time-scale image. Though more abstract in nature than
other time-frequency analysis methods, this process creates a pre-segmented image
that is well-suited for automated pattern recognition tasks. For some signals a direct
correlation can be made between particular features of these images and events in
the original signal, as we will demonstrate in the detection of subsurface flaws in
microelectronics using high-frequency contact ultrasound.

For most applications involving noisy real-world data, useful information remains
hidden in the transformed signal. In these situations we extend our analysis of the
binary fingerprint image by using image processing techniques to create many one-
dimensional parameter waveforms that describe our image (and by extension, the
original signal). In essence we have transformed the original one-dimensional signal
into a set of one-dimensional signals in a somewhat abstract domain, but one in which
the time information is preserved.

Trying to find patterns among all of these parameter waveforms requires a formal
framework. Statistical pattern classification provides this structure, and the basics of
the field are introduced in Chapter 3 through a toy problem. In particular we will
focus on the importance of feature selection to intelligently choose a set of features
that best describes the original signal. Creating a robust feature vector requires a large and varied set of training data, but if done correctly can create an information-dense system that can quickly classify any new signal.

In Chapter 4 we return to the issues encountered in attempting to create an autonomous walking-speed robot. In order to be useful in unstructured real-world environments, an autonomous robot must have a way to sense its surroundings and then act upon that information. This is best accomplished by collecting data from many complementary sensors to use in decision-making algorithms. Here we focus on the sensing issue by investigating the performance of different sensor modalities on our robotic platform, rMary. Both passive and active sensor – video, infrared, radar, and acoustic – are explored in outdoor environments, with an emphasis on identifying the capabilities and limitations of low-cost sensors such as the Microsoft Kinect.

Analysis of image data acquired from RGB or infrared video sensors is a well-studied problem with less new physics to exploit, leading us to focus on acoustic echolocation as a long-range sensor modality for mobile robotics. Chapter 5 describes acoustic echolocation measurements for the test case of detecting and classifying oncoming vehicles. A dataset is collected from both stationary and moving vehicles to improve detection methods and data quality.

Chapter 6 combines the DWFP feature extraction and statistical pattern classification framework described earlier to determine which type of vehicle is approaching the robotic platform at distances up to 50 m. Results are presented for several types of excitation signals and at different distances and show overall good performance.

To further understand the propagation of the acoustic signal and the physics of the scattering interaction between the signal and various vehicle targets, we use numerical simulations running on a distributed computing resource, described in Chapter 7.

Numerical solutions to the Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation
allow us to study the propagation and directionality of the nonlinear sound beam generated by the parametric array. Using visualizations from these simulations we can understand how the physical design of the parametric array affects the directionality of the sound beam and optimize the beam so that it interacts most strongly with the target.

The physics of the acoustic signal interacting with the target is modeled using the Acoustic Finite Integration (AFIT) technique. These are fully three-dimensional simulations in real-world sized spaces of several meters on a side, into which any three-dimensional model can be imported as a scattering target. Here, we use a parametric three-dimensional vehicle model to create representative models for five classes of vehicles and study the differences in scattering behavior between these classes.

These simulations provide both a visualization of the full, three-dimensional pressure field and the time-domain evolution of the pressure field at a given spatial point as a function of time. A unique impulse response method allows the calculation of the scattering field for long signals without the need for a massive simulation space.

Finally, in Chapter 8, we will look ahead at possible future directions in sensor fusion for mobile robotics.
Chapter 2

Finding information in noisy data

Many measurements acquired from physical systems are one-dimensional time domain signals, commonly representing amplitude as a function of time. In many cases useful information can be extracted from the signal directly. Using the waveform of an audio recording as an example, the total volume of the recording at any point in time is simply the amplitude of the signal at that time point. More in-depth analysis of the signal could show that regular, sharp, high-amplitude peaks are drum hits, while broader peaks are sustained organ notes. Amplitude, peak sharpness, and peak spacing are all examples of features that can be used to identify particular events occurring in the larger signal.

As signals become more complicated, either due to their innate nature (such as an audio recording featuring an entire orchestra as compared to a single instrument) or added noise, it becomes more difficult to identify particular features in the waveform and correlate them to physical events. Features that were previously used to differentiate signals no longer do so reliably.

In this chapter we will discuss how we can find new, useful features that describe our signal by creating representations of the original one-dimensional time domain
signal in alternative domains. As we will see, these transformations reveal useful
features hidden inside the original signal at the expense of a more abstract connection
to the original signal and increased computational requirements.

2.1 Time-frequency analysis of signals

2.1.1 Fourier transform

One of the most useful, and most common, transformations we can make on a time
domain signal is the conversion to a frequency-domain spectrum. For a real signal
\( f(x) \), this is accomplished with the Fourier transform

\[
\mathcal{F}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt.
\] (2.1)

The resultant signal \( \mathcal{F}(\omega) \) is in the frequency domain, with angular frequency \( \omega \)
related to the natural frequency \( \xi \) (with units cycles per second) by \( \omega = 2\pi \xi \). An
inverse Fourier transform will transform this signal back to the time domain. Since
this is the symmetric formulation of the transform, the inverse transform can be
written as

\[
f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathcal{F}(\omega) e^{i\omega t} d\omega.
\] (2.2)

Since the Fourier transform is just an extension of the Fourier series, looking at this
series is the best way to understand what actually happens in the Fourier transform.
The Fourier series, discovered in 1807, decomposes any periodic signal into a sum of
sines and cosines. This series can be expressed as the infinite sum

\[
f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nt) + b_n \sin(nt),
\] (2.3)
where the $a_n$ and $b_n$ are the Fourier coefficients. By finding the values of these coefficients that best describe the original signal, we are describing the signal in terms of some new basis functions, here, sines and cosines. The relation to the complex exponential given in the Fourier transform comes from Euler's formula, $e^{2\pi i \theta} = \cos 2\pi \theta + i \sin 2\pi \theta$.

In general, any continuous signal can be represented by a linear combination of orthonormal basis functions (specifically, the basis functions must define a Hilbert space). Sines and cosines fulfill this requirement and, because of their directly relevance to describing wave propagation, provide a physically relatable explanation for what exactly the decomposition does — it describes the frequency content of a signal.

In practice, since real-world signals are sampled from a continuous measurement, calculation of the Fourier transform is accomplished using a discrete Fourier transform. A number of stable, fast algorithms exist and are staples of any numerical signal processing analysis software. As long as the Nyquist-Shannon sampling theorem is respected (sampling rate $f_s$ must be at least twice the maximum frequency content present in the signal), no information about the original signal is lost.

### 2.1.2 Short-time Fourier transform

While the Fourier transform allows us to determine the frequency content of a signal, all time domain information is lost in the transformation. As an extension of our earlier example, the spectrum of the audio recording tells us which frequencies are present but not when those notes were being played.

The simple solution to this problem is to look at the Fourier transform over a series of short windows along the length of the signal. This is called the Short-Time
Fourier Transform (STFT), and implemented as

$$\text{STFT}\{f(t)\}(\tau, \omega) = \mathcal{F}(\tau, \omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) \bar{\omega}(t - \tau) e^{-i\omega t} dt,$$  \hspace{1cm} (2.4)$$

where $$\bar{\omega}(t - \tau)$$ is a windowing function that is nonzero for only a short time (typically a Hann window, described in the discrete domain by $$\bar{\omega}(n) = \sin^2 \left( \frac{n\pi}{N-1} \right)$$). Since this is an invertible process it is possible to recreate the original signal using an inverse transform, but windowing of the signal makes an inversion more difficult [1, 2].

Taking the squared magnitude of the STFT ($$|\mathcal{F}(\tau, \omega)|^2$$) and displaying the result as a color-mapped image with frequency on the vertical axis and time on the horizontal axis shows the evolution of the frequency spectrum as a function of time. These plots are referred to as spectrograms or waterfall plots, an example of which is shown in Figure 2.1.

![Figure 2.1: The spectrogram (bottom) of the William and Mary Alma Mater, performed by the William and Mary Chorus, provides information about the frequency content of the signal not present in the time domain waveform (top).](image)

It is important to note that this transformation from the one-dimensional time domain to a joint time-frequency domain creates a two-dimensional representation of the signal. Adding a dimension to the problem gives us more information about our signal at the expense of more difficult analysis.
The more serious limitation of the STFT comes from the uncertainty principle known as the Gabor limit,

\[ \Delta t \Delta \omega \geq \frac{1}{2}, \quad (2.5) \]

which basically says that a function cannot be both time and band limited. That is, it is impossible to simultaneously localize a function in both the time domain and the frequency domain. This leads to resolution issues for the STFT - a short window will provide precise temporal resolution and poor frequency resolution, while a wide window has the exact opposite effect [3].

### 2.1.3 Other methods of time-frequency analysis

The development of quantum mechanics in the mid-20th century created a number of alternative time-frequency representations because the mathematics are similar in both the position-momentum and time-frequency domains. One of these is the Wigner-Ville distribution first introduced in 1932, which maps the quantum mechanical wavefunction to a probability distribution in phase space. In 1948, Ville wrote a time-frequency formulation,

\[ W(\tau, \omega) = \int_{-\infty}^{\infty} f \left( \tau + \frac{t}{2} \right) f^* \left( \tau - \frac{t}{2} \right) e^{-i\omega t} dt, \quad (2.6) \]

where \( f^*(t) \) is the complex conjugate of \( f(t) \). This can be thought of as the Fourier transform of the autocorrelation of the original signal \( f(t) \), but because it is not a linear transform, cross terms occur when the input signal is not monochromatic [4].

Gabor also tried to improve the resolution issues with the STFT by introducing
the eponymous transform

\[ G(\tau, \omega) = \int_{-\infty}^{\infty} e^{-\pi(t-\tau)^2} e^{-i\omega t} f(t) \, dt, \]  

(2.7)

which is basically the STFT with a Gaussian window function. Like the STFT, this is a linear transformation and there is no problem with cross-terms. By combining the Wigner-Ville and Gabor transforms, we can mitigate the effects of the cross terms and improve the resolution of the time-frequency representation [5]. One possible representation of the Gabor-Wigner transform is

\[ D(\tau, \omega) = G(\tau, \omega) \times W(\tau, \omega). \]  

(2.8)

2.1.4 Wavelets

The overarching issue with any of the time-frequency methods previously discussed is that the basis of the Fourier transform is chosen with the assumption that the signals to be analyzed are periodic or infinite in time. Most real-world signals are not periodic but rather change a lot over time. This problem becomes even more clear when looking at finite signals with sharp discontinuities. Approximating such signals as linear combination of sinusoids creates overshoot at the discontinuities, the well-known Gibbs phenomenon illustrated in Figure 2.2 [6].

Instead we can use a basis of finite signals, called wavelets, to better approximate real-world signals. The wavelet transform is written as

\[ \mathcal{W}(\tau, s) = \frac{1}{\sqrt{s}} \int_{-\infty}^{\infty} f(t) \psi \left( \frac{t-\tau}{s} \right) dt. \]  

(2.9)

A comparison to the STFT (2.4) shows that this transform decomposes the signal
Figure 2.2: Attempting to approximate a square wave using Fourier components (sines and cosines) creates large oscillations near the discontinuities. Known as the Gibbs phenomenon, this overshoot increases with frequency (as more sums are added to the Fourier series) but eventually approaches a finite limit.

not into linear combinations of sines and cosines, but into linear combinations of wavelet functions $\psi(\tau, s)$. We can relate this to the Fourier decomposition (2.3) by defining the wavelet coefficients

$$c_{jk} = \mathcal{W}(k2^{-j}, 2^{-j}).$$

(2.10)

Here, $\tau = k2^{-j}$ and is referred to as the dyadic position and $s = 2^{-j}$ and is called the dyadic dilation. Basically, we are decomposing our signal in terms of a wavelet that can move (position $\tau$) and deform by stretching or shrinking (scale $s$). This
transforms our original signal into a joint time-scale domain, rather than a frequency domain (Fourier transform) or joint time-frequency domain (STFT). Although the wavelet transform doesn't provide any direct frequency information, scale is related to the inverse of frequency, with low scale decompositions relating to high frequency and vice-versa. This relationship is often exploited to de-noise data by removing information at particular scales (Figure 2.3).

![Diagram](image)

Figure 2.3: A signal $s(t)$ is decomposed into approximations ($A$) and details ($D$), corresponding to low and high-pass filters respectively. By continually decomposing the approximation coefficients in this manner and removing the first several levels of details, we have effectively applied a low-pass filter to the signal.

In addition to representing near-discontinuous signals better than the STFT, the dyadic (factor-of-two) decomposition of the wavelet transform allows an improvement in time resolution at high frequencies (Figure 2.4).

In the time domain, wavelets are completely described by the wavelet function (mother wavelet $\psi(t)$) and a scaling function (father wavelet $\phi(t)$). The scaling function is necessary because stretching the wavelet in the time domain reduces the bandwidth,
Figure 2.4: The STFT has similar time resolution at all frequencies, while the dyadic nature of the wavelet transform affords better time resolution at high frequencies (low scale values) requiring an infinite number of wavelets to accurately capture the entire spectrum. This is similar to Zeno's paradox, in which trying to get from point A to point B by crossing half the remaining distance each time is ultimately fruitless. The scaling function is an engineering solution to this problem, allowing us to get close enough for all practical purposes by covering the rest of the spectrum [7].

As discussed above, in order to completely represent a continuous signal, we must make sure that our wavelets form an orthonormal basis. Since as part of the decomposition we are allowed to scale and shift our original wavelet, we only need to ensure that the mother wavelet is continuously differentiable and compactly supported. For our analysis we will use the wavelet definitions and transform algorithms included in Matlab [8].

The Haar wavelet is the simplest example of a wavelet – a discontinuous step function with uniform scaling function. The Haar wavelet is also the first (db1) of the Daubechies family of wavelets abbreviated dbN, with order N (which is the number of vanishing moments). Historically, these were the first compactly supported orthornormal set of wavelets and were soon followed by Daubechies' slightly modified and least asymmetric Symlet family. The Coiflet family, also exhibiting vanishing moments, were also created by Daubechies at the request of other researchers [9].
The Meyer wavelet has both its scaling and wavelet functions defined in the frequency domain, but is not technically a wavelet because its wavelet function is not compactly supported. However, $\psi \to 0$ as $x \to \infty$ fast enough that the pseudo-wavelet is infinitely differentiable. This allows the existence of good approximations for use in discrete wavelet transforms, and we will consider the Meyer and related Discrete Meyer functions as wavelets for our analysis.

Both the Mexican hat and Morlet wavelets are explicitly defined and have no scaling function. The Mexican hat wavelet is proportional to the second derivative function of the Gaussian probability density function, while the Morlet wavelet is defined as $\psi(x) = Ce^{-x^2} \cos(5x)$, with scaling constant $C$.

The wavelets used are shown in Table 2.1, alongside their scaling and wavelet functions.
Table 2.1: A selection of wavelet families included in Matlab that were used as mother wavelets for continuous and/or discrete wavelet transforms. For wavelet families with multiple representations (i.e., Daubechies, symlets, and coiflets), the default used is shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Scaling function</th>
<th>Wavelet function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haar (haar)</td>
<td><img src="image" alt="Haar Scaling" /></td>
<td><img src="image" alt="Haar Wavelet" /></td>
</tr>
<tr>
<td>Daubechies (db3)</td>
<td><img src="image" alt="Db3 Scaling" /></td>
<td><img src="image" alt="Db3 Wavelet" /></td>
</tr>
<tr>
<td>Symlets (sym5)</td>
<td><img src="image" alt="Sym5 Scaling" /></td>
<td><img src="image" alt="Sym5 Wavelet" /></td>
</tr>
<tr>
<td>Coiflets (coif3)</td>
<td><img src="image" alt="Coif3 Scaling" /></td>
<td><img src="image" alt="Coif3 Wavelet" /></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Scaling function</th>
<th>Wavelet function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meyer (meyr)</td>
<td><img src="image1" alt="Meyer Scaling" /></td>
<td><img src="image2" alt="Meyer Wavelet" /></td>
</tr>
<tr>
<td>Discrete Meyer (dmey)</td>
<td><img src="image3" alt="Discrete Meyer Scaling" /></td>
<td><img src="image4" alt="Discrete Meyer Wavelet" /></td>
</tr>
<tr>
<td>Mexican hat (mexh)</td>
<td>Explicit</td>
<td><img src="image5" alt="Mexican Hat Scaling" /></td>
</tr>
<tr>
<td>Morlet (morl)</td>
<td>Explicit</td>
<td><img src="image7" alt="Morlet Scaling" /></td>
</tr>
</tbody>
</table>
2.2 The Dynamic Wavelet Fingerprint (DWFP)

So far we have seen both the benefits and difficulties present in several different methods of time-frequency analysis. While alternative time-frequency transformations can improve the resolution limits of the STFT, they create their own problems such as the cross-term in the Wigner-Ville transform. Combinations of transforms can reduce these effects while still offering increased resolution, but this comes at the cost of computational complexity. Wavelets offer an alternative basis for decomposition that is more suited to finite real-world signals, but without the direct relationship to frequency.

One of the yet undiscussed issues with time-frequency representations of signals is the added complexity of the resultant time-frequency images. Just as displaying a one-dimensional signal requires a two-dimensional image, viewing a two-dimensional signal requires a three-dimensional visualization method. Common techniques include three-dimensional surface plots that can be rotated on a computer screen or colormapped two-dimensional images where the value at each point is mapped to a color.

While these visualizations work well for human interpretation of the images, computers have a difficult time distinguishing between those parts of the image we care about and those that are just background. This difficulty with image segmentation is especially true for noisy signals. The human visual system is evolutionarily adapted to be quite good at this (in previous times, detecting a lion hiding in the tall grass was a matter of life or death), but computers lack such an advantage. Current automated image segmentation methods work well for scenes where a single object is moving in a predictable path across a mostly stationary background [10, 11, 12].

We will instead use an alternative time-frequency representation called the Dynamic Wavelet Fingerprint (DWFP) that has been useful in past work to reveal subtle
features in noisy signals [13, 14, 15, 16, 17]. This technique takes a one-dimensional
time domain waveform and converts it to a two-dimensional time-scale image, with
the main advantage of generating a pre-segmented binary image that can be analyzed
using existing image processing techniques.

2.2.1 Feature creation

The DWFP process, illustrated in Figure 2.5, first filters a one-dimensional signal
using a stationary discrete wavelet transform. This decomposes the signal into wavelet
components at a set number of levels, removes the chosen details, and then uses the
inverse stationary wavelet transform to recompose the signal. The number of levels,
details to remove, and wavelet used for the transform are all user parameters. A
Tukey window can also be applied to the filtered signal at this point to smooth out
behavior at the edges.

Next, the wavelet coefficients are created using a continuous wavelet transform.
The normalized coefficients form a three-dimensional surface, and can be thought of as
'peaks' or 'valleys' depending on if the coefficients are positive or negative. Slicing this
surface (both slice thickness and number of slices are user parameters) and projecting
the slices to a plane generates a two-dimensional binary image. The vertical axis of
this image is scale (inversely related to frequency), and the horizontal axis remains
time, allowing direct comparison to the original one-dimensional signal.

The image often resembles a set of fingerprints (hence the name), but most im-
portantly the image is pre-segmented and can be easily analyzed by standard image
processing techniques. Since the slicing process does not distinguish between peak
(positive coefficients) and valleys (negative coefficients) we can instead do the slicing
operation in two steps, keeping the peak and valley projections separate. This gener-
ates two fingerprint images for each signal – one for peaks and one for valleys – which
Figure 2.5: The dynamic wavelet fingerprint process (DWFP) uses a continuous wavelet transform to map a one-dimensional signal (a) to a three-dimensional surface of wavelet coefficients (b). Slicing this surface (c) and projecting the slices to a plane creates a two-dimensional time-scale image that often resembles a set of individual fingerprints (d). (Image modified and used by permission from [18])

A number of additional features can be extracted from this fingerprint image. Some of the features we extract are by definition continuous, for example, a simple count of the number of ridges at each time point. However, many of the features that we want to extract from the image are tied to a particular individual fingerprint, requiring us to first identify and consecutively label the individual fingerprints. We use a measure of nearly connectedness, in which pixels of the same value within a set distance of each other are considered connected, to label each individual fingerprint. This measure works well as long as each individual fingerprint is spatially separated from its neighbor, something that is not necessarily true for the ternary fingerprint.
images. For those cases we actually decompose the ternary image into two separate binary images, label each one individually, and then recombine and relabel the two images (Figure 2.6).

Figure 2.6: To consecutively label the individual fingerprints within the fingerprint image, the valleys (top left) and peaks (top right) images are first labeled individually and then combined into an overall labeled image (bottom).

In some cases, the automated labeling will classify objects as a fingerprint even though they may not represent our view of a fingerprint. While this won’t affect the end results because such fingerprints won’t contain any useful information, it can slow down an already computationally-intensive process. To reduce these false fingerprints, an option is added to restrict the allowed solidity range for an object to be classified as an individual fingerprint.
2.2.2 Feature extraction

Now that the location and extent of each individual fingerprint has been determined, we apply standard image processing libraries included in Matlab to extract features from the image [19]. The resemblance of our images to fingerprints, for which a large image recognition literature already exists, can be exploited in this process [20, 21].

These parameter waveforms are then linearly interpolated to facilitate a direct comparison to the original time domain signal. In the end, approximately 25 one-dimensional parameter waveforms are created for each individual measurement. Some of these features are explained in more detail in Table 2.2.

The user has control of a large number of parameters in the DWFP creation and feature extraction process (Table 2.3), which greatly affect the appearance of the fingerprint images, and thus the extracted features. There is no way to tell a priori which combination of parameters will create the ideal representation for a particular application. Past experience with analysis of DWFP images helps us to avoid an entirely brute force implementation for many applications. However, in some cases the signals to be analyzed are so noisy that humans are incapable of picking out useful patterns in the fingerprint images. For these applications we use the formal language of pattern classification (further discussed in Chapter 3) and a computing cluster to run this feature extraction process in parallel for a large number of parameter combinations.
Table 2.2: A number of features are extracted from both the raw signal and the wavelet fingerprint image using the Matlab image processing toolbox `regionprops` analysis to create an optimized feature vector for each instance.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signal features</strong></td>
<td></td>
</tr>
<tr>
<td>Raw signal</td>
<td>Original signal</td>
</tr>
<tr>
<td>Filtered signal</td>
<td>Wavelet filtered original signal</td>
</tr>
<tr>
<td>Raw PSD</td>
<td>Power Spectral Density of raw signal</td>
</tr>
<tr>
<td>Filtered PSD</td>
<td>Power Spectral Density of filtered signal</td>
</tr>
<tr>
<td><strong>Fingerprint image features</strong></td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>Number of pixels in the region</td>
</tr>
<tr>
<td>Filled Area</td>
<td>Number of pixels in the bounding box (smallest rectangle that completely encloses the region)</td>
</tr>
<tr>
<td>Extent</td>
<td>Ratio of pixels in the region to pixels in bounding box, calculated as ( \frac{\text{Area}}{\text{Area of bounding box}} )</td>
</tr>
<tr>
<td>Convex Area</td>
<td>Area of the convex hull (the smallest convex polygon that contains the area)</td>
</tr>
<tr>
<td>Equivalent Diameter</td>
<td>Diameter of a circle with the same area as the region, calculated as ( \sqrt{\frac{4\pi \text{Area}}{\pi}} )</td>
</tr>
<tr>
<td>Solidity</td>
<td>Proportion of pixels in the convex hull to those also in the region, calculated as ( \frac{\text{Area}}{\text{Convex Area}} )</td>
</tr>
<tr>
<td>xCentroid</td>
<td>Center of mass of the region along the horizontal axis</td>
</tr>
<tr>
<td>yCentroid</td>
<td>Center of mass of the region along the vertical axis</td>
</tr>
<tr>
<td>Major Axis Length</td>
<td>Pixel length of the major axis of the ellipse that has the same normalized second central moments as the region</td>
</tr>
<tr>
<td>Minor Axis Length</td>
<td>Pixel length of the minor axis of the ellipse that has the same normalized second central moments as the region</td>
</tr>
<tr>
<td>Eccentricity</td>
<td>Eccentricity of the ellipse that has the same normalized second central moments as the region, computed as the ratio of the distance between the foci of the ellipse and its major axis length</td>
</tr>
<tr>
<td>Orientation</td>
<td>Angle (in degrees) between the x-axis and the major axis of the ellipse that has the same second-moments as the region</td>
</tr>
<tr>
<td>Euler Number</td>
<td>Number of objects in the region minus the number of holes in those objects, calculated using 8-connectivity</td>
</tr>
<tr>
<td>Ridge count</td>
<td>Number of ridges in the fingerprint image, calculated by looking at the number of transitions between pixels on and off at each point in time</td>
</tr>
</tbody>
</table>
Table 2.3: List of user parameters in DWFP creation and feature extraction process.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavelet filtering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filtmeth</td>
<td>filt, filtandwindow, window, none</td>
<td>how to filter data</td>
</tr>
<tr>
<td>wvtpf</td>
<td>(wavelet name)</td>
<td>filtering wavelet</td>
</tr>
<tr>
<td>numlvls</td>
<td>$Z^+$</td>
<td>number of levels to filter</td>
</tr>
<tr>
<td>swdtoremove</td>
<td>$[Z^+]$</td>
<td>details to remove</td>
</tr>
<tr>
<td>Wavelet transform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wvt</td>
<td>(wavelet name)</td>
<td>transform wavelet</td>
</tr>
<tr>
<td>ns</td>
<td>$Z^+$</td>
<td>number of scales for transform</td>
</tr>
<tr>
<td>normconstant</td>
<td>$Z^+$</td>
<td>normalization constant</td>
</tr>
<tr>
<td>numslclices</td>
<td>$Z^+$</td>
<td>number of slices</td>
</tr>
<tr>
<td>slicethickness</td>
<td>$R^+$</td>
<td>thickness of each slice</td>
</tr>
<tr>
<td>Feature extraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>saveimages</td>
<td>binary switch</td>
<td>save fingerprint images?</td>
</tr>
<tr>
<td>fullorred</td>
<td>full, reduced</td>
<td>require certain solidity</td>
</tr>
<tr>
<td>solidity_range</td>
<td>$[R \in [0,1], R \in [0,1]]$</td>
<td>allowable solidity range</td>
</tr>
</tbody>
</table>

2.3 An application: Ultrasonic detection of flaws in microelectronics

In order to better illustrate the DWFP, we discuss an application using high-frequency pulse-echo ultrasound to detect subsurface flaws in microelectronic components. In particular, suspected counterfeit router components need to be inspected quickly and nondestructively with portable equipment that doesn't require extensive training to operate. Although ultrasound in the 100 MHz frequency range routinely images the relevant subsurface features in microelectronic components, scanning acoustic microscopes are neither portable nor inexpensive, and they require the component to be submerged in a bath of coupling water. Our alternative approach uses a custom-designed ultrasonic probe to directly contact the component surface without requiring
submersion in water. Instead of using computer-controlled scanning to produce images, the pulse-echo waveforms are processed using the DWFP to find differences in the waveforms that indicate flaws in subsurface layers and interfaces of the part.

2.3.1 Background

Counterfeit, recycled, and maliciously modified integrated circuits have increasingly become a threat to the United States' information technology infrastructure. Though several industry associations have taken measures to limit counterfeiting, some industry experts estimate the cost at $100 - $200 billion annually, which represents nearly 10% of all electronic equipment sold worldwide [22]. In particular, the sale of hard-to-get and obsolete parts, often non-ROHS (restriction-of-hazardous-substances) compliant, has seen an increase in counterfeiting. These components have a limited marketability but are still common and vital components in the aerospace, medical, and military industries [23].

2.3.2 Ultrasonic measurements of microelectronics

Ultrasound has been used to create images of the inside of objects from the mid-1900s. While ultrasound imaging had its origins in the medical profession as an alternative to invasive procedures or ionizing imaging technologies such as x-rays or CT scans, such techniques have since been applied to a variety of non-medical problems in such fields as materials science and non-destructive testing [24]. Coupling the ultrasonic energy into the test piece often turns out to be the main issue.

The current industry standard for ultrasonic imaging of microelectronic components uses a process called acoustic microscopy, in which the component is submerged in water and scanned over a well-defined grid to create an image [25, 26]. This process
is able to create highly detailed images of the internal structure of small objects using a high-frequency ultrasonic transducer and a computer-controlled scanner. The images acquired from ultrasonic scans, shown in Figure 2.7 for one of our microelectronic components, are standardized across the industry. An A-scan is simply the waveform at one particular point, with time (which relates to distance) on the x-axis and amplitude on the y-axis. A B-scan provides a profile (cross-section) view by taking a row of A-scans and converting the amplitude at each point in time to a brightness or color value. The C-scan provides a plan view of the entire piece by looking at the maximum value of the A-scan over a time gate, which can be adjusted to view a particular depth.

This method requires the microelectronics be submerged in a water bath. In addition, the special apparatus, training, and time required to scan a single component prevents this from being a feasible method to quickly detect counterfeit components. Instead we will focus on using contact ultrasound, where a small amount of glycerin or a gel pad is used for coupling. Since quick detection of a counterfeit component is the goal rather than creating a C-scan image, we will focus on acquiring and processing A-scan measurements.

The first task was to design, test, and optimize a handpiece which contains a high-frequency (up to 100 MHz) ultrasound transducer that couples the ultrasound signal into and back out of the chip surface in pulse-echo mode. This requires finding a balance between the ease-of-use ergonomics and the fundamental requirement of recording high quality data. The necessary high-frequency contact transducers are a semi-custom item requiring specialized expertise and were procured from Olympus NDT (Panametrics). Their V2012 (BC) is a 100 MHz, 1/8 in. diameter element transducer with an integral 2.596 ms fused silica delay line. Lower frequency units are also available in the same case size.
Figure 2.7: The three different visualizations of 50MHz submerged ultrasound scans show the internal components of an integrated circuit at a particular point (A-scan), in a cross-sectional view (B-scan), and over the entire component (C-scan).
Next we wish to reduce the effective contact footprint of these \(1/8''\) diameter transducers to increase the spatial resolution of the measurement. At low-megahertz frequencies this is routinely done via a tapered delay line made from metal or stiff plastic, but such materials attenuate too strongly at 100 MHz. Instead we machined quartz delay lines, which is the same material as the transducer’s internal delay line. Two shapes were explored: a narrow rod and a tapered, conical delay line, with final contact diameters of 1.58 mm, shown in Figure 2.8. The greater contact area with the transducer and the less-than-expected effect of internal reflections led us to use the tapered delay line exclusively.

Combined, the entire transducer assembly consists of a transducer, tapered delay line, and necessary assembly hardware. The tapered delay line is held against the face of the integral delay line by an attachment collar, carefully adjusted to not fracture the delicate quartz delay line. Absolute cleanliness of the two quartz surfaces is required, and a very small droplet of glycerin provides acoustic coupling between the two delay lines.

Figure 2.8: Both straight and conical quartz delay lines attach directly to the transducer via an oil couplant and reduce the footprint of the transducer from \(1/8''\) to 1.58 mm. In our tests the conical delay line worked best with few problems from internal reflections.
Because of the small contact area of the tip and the sensitivity of its perpendicularity on the measured waveforms, it would be difficult to obtain repeatable waveforms free-hand. A stabilization device (Figure 2.9) helps hold the transducer in the correct orientation while still allowing for easy movement to collect data across the component's surface. Coupling between the transducer assembly and the sample was accomplished with small amounts of a standard ultrasound oil couplant.

![Image](image_url)

Figure 2.9: The base of the stabilizer arm is placed next to an open router to allow inspection of any component while keeping the transducer in the correct orientation for optimal coupling.

The final step in creating our benchtop ultrasound apparatus is driving the transducer assembly and recording the measured signal. An industrial computer with ICA-compatible backplane holds the ultrasonic pulser/receiver (Matec SR-9000) and A/D (Gage CompuScope 2125) boards. We chose a spike pulser/receiver with a 100
MHz bandwidth to drive the transducer and amplify the return signal, requiring at least an 8-bit digitizer card with a bandwidth of 200 MHz. Connections between the cards are control triggering and data acquisition and use standard BNC cabling, as does the connection from the pulser/receiver to the transducer assembly (Figure 2.10).

![Figure 2.10: Schematic of the connections necessary to acquire ultrasonic measurements from microelectronic components.](image)

Software drivers included with the Gage and Matec cards provide high-level access to the hardware, allowing us to change hardware settings from inside a Matlab GUI. This interface provides an easy-to-use data capture system that can be operated by a technician-level operator (Figure 2.11).

### 2.3.3 Creating flawed microelectronics samples

In order to test the experimental apparatus and develop the signal processing analysis required to automatically classify components as flawed or unflawed, we acquired a set of unflawed chips and introduced delaminations into several of them.

We are interested in the type of flaws most often found in recycled components. In this flaw formation process moisture is absorbed in the hygroscopic epoxy encapsulant,
leading to reduced adhesion. Differential expansion during the solder reflow process, exacerbated by the higher temperatures required by the new lead-free solder alloys, creates delaminations. Moisture then diffuses into these voids, where further expansion and cracking occurs [27]. This is often referred to as the ‘popcorn effect’ and is divided into three modes according to the location of the flaws [28]:

Mode I: cracks occurring at the underside of the die pad (most common)

Mode II: cracks occurring in the adhesive at the die pad interface

Mode III: cracks occurring at the chip surface (rare, but easiest to detect).
Surface mount packages are more susceptible to these effects because they present more surface area for moisture absorption, and most of the router components are in surface mount packages. We introduced flaws in a total of 19 components, split between the Thin-Shrink Small Outline Package (TSSOP, 11 components) and Plastic Leaded Chip Carrier (PLCC, 8 components) package styles.

The IPC has established a classification for moisture sensitivity and an experimental process for testing, which was used to create flawed components [29]. We assume that the most aggressive environment outlined in this standard, one week at 85°C and 85% relative humidity, is most likely to introduce flaws. A laboratory oven modified with a nebulizer is able to create this environment, and can be monitored by remote probe.

Between 15 minutes and 4 hours after removal from the moisture soak, the components are exposed to reflow conditions. Generally this reflow process is performed in a conveyor-fed oven containing an infrared heat source, but rather than purchase a large and expensive reflow oven for this simple task, we simulated this type of fast heating via a hot-air SMT rework station which duplicated the 'soldering shock' that occurs in the flaw formation process. A closed-loop controller regulated the lamp power to simulate a typical reflow heating profile in which the temperature reached 260°C. A very small thermocouple embedded within the test package confirms the achievement of these desired internal temperatures.

After the components were exposed to the moisture soak and reflow process, visible Mode III flaws were confirmed in four of these packages, all of the thinner TSSOP form factor. Since this is the rarest, but most easily detectable, flaw mode, we had confidence that flaws of other modes were present in other components. This would later be confirmed with standard ultrasound immersion scans, after contact ultrasound measurements were acquired.
The previously discussed benchtop ultrasound apparatus was used to collect data from each of these 19 components, as well as from 6 control packages, 3 each of the TSSOP and PLCC package styles. For each component, A-scan waveforms were acquired at approximately 30-48 locations in a rough grid pattern across the face of the component. In this way we can exploit positional information to better understand features that may be present in different regions of the component without the hassle of an automated high-precision scanner that would be necessary to create an image. To prevent operator bias, multiple measurements of each chip were taken. In total, 1710 different waveforms were collected, at frequencies of 100, 75, and 50 MHz.

2.3.4 DWFP analysis

By their nature, waveforms recorded from contact ultrasound measurements are much noisier than those recorded in submerged ultrasound measurements. Careful design of the delay line can help separate those reflections we are most concerned about, but also creates more noise and false peaks from internal reflections. Finding the reflections we care about in these measurements by looking at the time-domain waveform is difficult, even for simple, homogeneous structures. For such complicated structures as microelectronics, and at such high frequencies where attenuation plays as important role, it is near impossible.

In order to analyze these waveforms and find features that correspond to internal structures of the components, we will use the time-frequency based DWFP analysis described above.

For ease of use, a Matlab GUI is used to create and view these fingerprint images, shown in Figure 2.12. The settings used in this process (such as the mother wavelet, number of slices, and slice thickness) can be easily changed in order to alter the appearance of the fingerprint image. This also allows users to view the fingerprint
images to try and find patterns that correspond to internal structures and flaws within the components. The human visual system is well adapted for this task, and the more eyes on the problem, the more likely a set of optimized features will be found to best characterize the components.

Figure 2.12: A custom Matlab GUI was built to assist in the DWFP analysis. The raw A-scan waveform (top) is filtered and windowed to focus on the region of interest (middle), and the wavelet fingerprint is constructed (bottom) based on a number of user-selectable parameters.

After systematic investigation of both the 100 and 75 MHz data acquired for both flawed and unflawed components, we concluded that because the delaminations which we introduced were all on the back side of the silicon, we were unable to identify obvious patterns that were useful for distinguishing flawed chips from unflawed chips. While higher frequency gives better spatial resolution, it also attenuates much more strongly, since attenuation is proportional to the square of frequency. This is especially an issue for the case of the component packages, which are mostly made of a plastic composite material that is less dense than metals and has a higher attenuation [30].

More sophisticated pattern classification approaches originally developed for other applications, such as the acoustic echolocation vehicle classification described in Chap-
ter 6, may be able to construct a more optimized feature vector that could better discriminate between flawed and unflawed components. However, the microelectronics dataset was too small to have sufficient statistics in the application of such methods.

Analysis of the 50 MHz data showed clear features indicating the component's silicon layer, as well as features indicating delaminations distal to the silicon. Figure 2.13 shows three typical 50 MHz fingerprints from unflawed and flawed components, with an obvious white feature that is absent when a delamination is present.

Now that a feature which indicates a delamination has been discovered, automated image processing routines can quickly look for this feature in each of the 30-48 measurement locations on a single component, making a binary flaw/no flaw decision at each point. Arranging these binary decisions according to the rough position where the measurements were acquired helps prevent false-positives. Contiguous regions of a positive flaw decision indicate a flawed component, which was verified with submerged C-scan images (Figure 2.14).

We have demonstrated the feasibility of using high-frequency contact ultrasound to acquire pulse-echo waveforms from microelectronics. These waveforms are processed with the DWFP algorithms to create fingerprint images in which the presence or absence of certain features correspond to the presence or absence of flaws. Once identifying features have been discovered, computer algorithms can be written to allow the computer return a 'flawed' or 'unflawed' decision. When such binary decisions are arranged roughly corresponding to each waveform's position on a component it becomes easy to determine whether a chip is flawed or not based on the presence of contiguous regions of flawed waveforms.
Figure 2.13: Comparison of the fingerprint images for three flawed (top) and three unflawed (bottom) components acquired at 50 MHz highlight differences in the signals not visible in the raw waveform. Since the horizontal axis is time delay of echoes (proportional to depth), we can identify the large feature with many ridges as the silicon layer and look for features that indicate a delamination immediately distal to this layer (red arrow). Here, the absence a stable white left-inclined oval indicates a delamination.
Figure 2.14: An image processing routine looks for the particular feature indicating a delamination in each of the 48 measurement locations on a single component. Contiguous regions of a flaw decision (right) indicate a flawed component, confirmed with submerged C-scan measurements (left).
2.3.5 Application to analysis of other signals

Here a visual inspection of the fingerprint images highlighted features that were able to describe a microelectronic component as 'flawed' or 'unflawed'. While this is a useful illustration of the DWFP process, it is important to discuss the limitations of the proof-of-concept analysis shown here.

The contact ultrasound measurements of microelectronics were acquired for a small set of samples in a well-controlled laboratory environment. Samples were artificially simplistic by design, limiting naturally occurring variations such as the type of packaging or chip size. Artificially introduced flaws always created delaminations distal to the silicon, providing a known time window in the measured signal in which to look for features indicating the presence of a flaw. This, combined with the relatively small amount of data, allowed human analysis of the fingerprint images to find stable features.

The main limitation of this analysis is that a 'flawed/unflawed' decision is a binary classification task performed individually at each measurement location. The entire chip can be classified as 'flawed/unflawed' only after measurements at multiple locations show the spatial extent of a flaw.

For comparison, DWFP analysis of real world acoustic echolocation data, which will be discussed more in Chapter 6, lacks many of these simplifying assumptions. Data is collected over a long period of time in noisy, unstructured environments while large variation between signals means that even determining which part of the signal contains useful information is not a trivial task. Likewise, aligning measurements from different vehicles so that the reflections align in time is a critical requirement. This is especially true because the massive amount of data required to form a robust data set makes human analysis of the fingerprint images impractical.

The most important difference is that the vehicle data needs to be classified into...
one of five possible classes, rather than the binary classification of microelectronics data as 'flawed/unflawed'. As shown in Chapter 6, extension of the DWFP feature extraction process allows these types of classifications with an accuracy of 94%.
Chapter 3

Statistical pattern classification

In many cases, signals from different objects cannot be distinguished from each other by visual inspection, even when the signal is represented in alternative domains. In these cases we can make use of machine learning algorithms that allow computers to analyze and classify our data.

Machine learning is a broad term for the process that allow a computer to analyze raw data and group the data into categories based on patterns within the raw data. This is similar to how humans would classify data into groups by finding specific features that are present in some classes but not in others. Even though computers lack the human visual system's eons of evolutionary insight into pattern classification, they make up for it by recognizing complex patterns in high-dimensional spaces which humans can't visualize.

Machine learning can be separated into two categories: supervised and unsupervised learning. In supervised learning, labeled data exists and the goal is to classify new data into one of these existing categories. In essence, we are deciding where to draw the decision boundary between classes in multidimensional space. In unsupervised learning, labeled data is not available, and in many cases it's not even clear how
many classes there should be. The task here is to place similar objects in the same
class and determining how to best cluster the data.

A number of texts have been written on machine learning [31, 32, 33, 34, 35]. A
few selected applications of pattern classification using the DWFP process include
detecting roof fall events in mines [17], determining the extent of periodontal disease
in teeth and flaw depth in metals with non-contact methods [36, 37], and detecting
counterfeit RFID (Radio-Frequency ID) tags [38, 18].

Our focus will be on supervised learning. In the materials that follow, pattern
classification will be understood to refer to supervised learning, and will be used
interchangeably with the terms pattern recognition and machine learning. A flowchart
describing the pattern classification process is shown in Figure 3.1.

3.1 Statistical pattern classification by example:
   analysis of glass

We next illustrate the pattern classification process with a simple example. A
number of such examples exist, from Fisher’s original iris dataset from 1936 that
uses 4 taxonomic measurements to differentiate between species of iris [39] to a 13
parameter chemical analysis to determine the origin of Italian wines [40]. These
are small datasets (150 irises and 178 wines, respectively) that are well suited for
visualization, but more complicated datasets are also freely available, such as the
nearly 49,000 measurement census income dataset that uses 14 factors to determine if
an individual has a greater than $50,000 annual income [41]. Over 245 such datasets
have been compiled and placed online by the University of California Irvine to allow
easy testing of improvements to pattern classification algorithms on standard, real-
world datasets [42].
Figure 3.1: A flowchart of the pattern classification process focuses on supervised machine learning where the classes are known. We emphasize the preprocessing steps to create an information dense feature vector to allow for quick and accurate classified by existing routines.
For this example we will use a dataset of different types of glass, based on their physical properties [43]. This data consists of 214 total samples divided into 6 classes: float-processed building windows, building windows, float-processed vehicle windows, containers, tableware, and headlamps (Table 3.1). Each of these samples is described by 9 physical measurements: the oxide content (by weight percent) of sodium (Na), magnesium (Mg), aluminum (Al), silicon (Si), potassium (K), calcium (Ca), Barium (Ba), and Iron (Fe), as well as the refractive index (n).

Table 3.1: Class distribution for the glass dataset.

<table>
<thead>
<tr>
<th>Major class</th>
<th>Minor class</th>
<th># Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>Float</td>
<td></td>
</tr>
<tr>
<td>Building</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>Vehicle</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Non-float</td>
<td>Building</td>
<td>76</td>
</tr>
<tr>
<td>Non-window</td>
<td>Container</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Tableware</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Headlamps</td>
<td>29</td>
</tr>
</tbody>
</table>

We can investigate several different relationships in our classification attempts of this data based on physical knowledge of the system, ranging from broad classifications (window vs. non-window) to specific (into which of these 6 classes does a new sample best fit). We would expect that the more specific the classification, the less accurate the performance. For explanatory purposes, we will focus on the broadest possible classification (window vs. non-window) and the classification of the non-window glass into sub-classes (container, tableware, or headlamp) but also look at why the most specific classification fails.
Looking at a small set of observations shown in Table 3.2, we can see that trying to classify a sample by a single measurement is impossible — for example, three different types of glass have no measurable iron (Fe) content.

Table 3.2: Measured values from several of the glass samples include the refractive index $n$ and weight percent content of various oxides (BWF: building windows float processed, BWNF: building windows non-float processed). Each row of this table is single observation described by a 9-dimensional feature vector $\mathbf{x}_i$.

<table>
<thead>
<tr>
<th>Label</th>
<th>$i$</th>
<th>$n$</th>
<th>Na</th>
<th>Mg</th>
<th>Al</th>
<th>Si</th>
<th>K</th>
<th>Ca</th>
<th>Ba</th>
<th>Fe</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWF</td>
<td>52</td>
<td>1.51926</td>
<td>13.20</td>
<td>3.33</td>
<td>1.28</td>
<td>72.36</td>
<td>0.60</td>
<td>9.14</td>
<td>0.00</td>
<td>0.11</td>
</tr>
<tr>
<td>BWNF</td>
<td>116</td>
<td>1.51846</td>
<td>13.41</td>
<td>3.89</td>
<td>1.33</td>
<td>72.38</td>
<td>0.51</td>
<td>8.28</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>BWNF</td>
<td>126</td>
<td>1.51872</td>
<td>12.93</td>
<td>3.66</td>
<td>1.56</td>
<td>72.51</td>
<td>0.58</td>
<td>8.55</td>
<td>0.00</td>
<td>0.12</td>
</tr>
<tr>
<td>Container</td>
<td>167</td>
<td>1.52151</td>
<td>11.03</td>
<td>1.71</td>
<td>1.56</td>
<td>73.44</td>
<td>0.58</td>
<td>11.62</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Headlamp</td>
<td>203</td>
<td>1.51514</td>
<td>14.85</td>
<td>0.00</td>
<td>2.42</td>
<td>73.72</td>
<td>0.00</td>
<td>8.39</td>
<td>0.56</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Instead, we need to look at more than one of these measured values to tell the different types of glass apart. Figure 3.2 shows a scatter plot of two features, weight percent sodium and refractive index, for the subset of data from non-window glass. The goal is to separate this plot into regions that clearly separate the different classes of data. New data can then be classified based on where its features lie compared to the decision boundary.

However, we see that there is a lot of overlap between the different classes of data. While it would be possible to draw a line (construct a decision boundary) that separates the different classes, such a boundary is highly-tuned and would not work well at classifying new data. Instead we strive to create a more generalized decision boundary that is able to accurately classify novel data. To do this, we add another measurement to our comparison, so that each observation is described by three measured values (Figure 3.3).
Figure 3.2: Clear decision boundaries cannot be drawn on a two-dimensional plot of the weight percent sodium (Na) vs. refractive index (n) for the three non-window classes of samples.

This process can be extended to as many dimensions as we have features, though we can't easily visualize these higher dimensions. Just as the decision boundary changed from a line in two-dimensions to a surface in three-dimensions, decision boundaries in higher-dimensional feature spaces will be hyperplanes. To facilitate discussion of this multidimensional feature space we introduce the following notation:

- Each set of measurements for a single sample (which we will call an observation or instance) is described by a feature vector $\mathbf{x}_i$ with observation index $i = 1, 2, \ldots, s$ for a total of $s$ observations (here we have 214 observations).

- These feature vectors describe each observation as a point in a multidimensional
Figure 3.3: Adding the weight percent magnesium measurements to a two-dimensional plot of the weight percent sodium (Na) vs. refractive index (n) for the three non-window classes of samples increases the dimensionality and allows for clearer, simpler decision boundaries (which are now surfaces rather than lines).

feature space, where the dimension of the space is equal to the length of the feature vector (here we have a 9-dimensional space).

- The goal of pattern classification is to map each feature vector to a class $\omega_k$, where index $k = 1, 2, \ldots$, total number of classes (here there are six known classes).

Obviously the dimension of the feature space cannot be larger than the number of features, but other interesting problems reveal themselves in high-dimensional spaces. As the number of dimensions increase, the volume of the possible feature space increases even faster. This causes our finite-length training data to become sparse.
and can lead to decreased classification performance. This is known as the ‘curse of dimensionality’, a term first coined by Bellman in 1961 in the field of dynamic optimization and applied to pattern recognition in 1968 by Hughes [44, 45]. The general rule of thumb is to reduce our feature space to no more than half the amount of available data, which is the minimum amount required to completely separate random data [46]. In cases where we have more knowledge about our data (most pattern classification situations), we can surpass this limit by incorporating correct knowledge about our data [31].

3.2 Feature extraction and selection

The first step in pattern classification is to create a small-dimension, information-dense feature vector that compactly represents our original signal \((x_j \text{ where } j < i)\). Because of the immense computing power available to the modern-day researcher, this step is all too often bypassed in favor of throwing all available raw data at the classifier. While this may still return favorable classification results, it is not the most efficient approach, and often the classification step can take hours or even days to make a single decision. Reducing the signal to its essentials, without discarding useful information, allows classification routines to run in seconds. This is important for real-time classification systems such as those necessary to create an autonomous robot. Reduction of our feature space through feature selection is also one of the ways that we can avoid the curse of dimensionality.

One popular method to reduce the dimension of the feature space is principal component analysis (PCA), first introduced by Pearson in 1901 and also called the Karhunen-Loève transform [47, 48]. In this process a random vector \(x \in \mathbb{R}^N\) can be
optimally re-written\(^1\) in terms of linear combinations of \(M < N\) independent vectors by projecting \(\mathbf{x}\) onto the eigenvectors \(\varphi_i\) corresponding to the largest eigenvalue \(\lambda_i\) of the covariance matrix \(\Sigma_x\). Essentially, PCA performs a coordinate transformation that rotates the axes to align with directions of maximum variance. This is often a useful property, but since there is no guarantee that the directions of maximum variance will provide the best features to discriminate between classes (and because no information about class labels is taken into account when performing the PCA), it is less than optimally designed for classification purposes.

Linear discriminant analysis (LDA)\(^2\) is more useful in pattern classification applications precisely because it takes the class labels into account when determining onto which projection vector \(\mathbf{x}\) should be mapped \([32, 49, 35]\).

To understand how LDA works, we will first consider the case with \(k\) classes \(\omega_k\). For each of these classes, a mean feature vector can be created as

\[
\mu_k = \frac{1}{N} \sum_{\mathbf{x} \in \omega_k} \mathbf{x}.
\]  

(3.1)

The simplest measure of discrimination between the classes is the difference of the class means, which can be thought of as the distance between them in our high-dimensional space\(^3\). However, this measure fails to account for the variance within a class. A better measure, and the basis of LDA, is to use some measure of the intraclass variance to normalize the interclass variance. Using the standard deviation of the mean as a measure of intraclass variance

\[
\sigma_k = \sqrt{\frac{1}{N} \sum_i (x_i - \mu_k)^2} \quad \text{where} \, \mathbf{x} \in \omega_k,
\]  

(3.2)

---

\(^1\)Minimizing the sum-square magnitude of the approximation error.

\(^2\)Interchangeably called Fisher’s linear discriminant.

\(^3\)For a simple, two-class problem this would be represented as \(|\mu_1 - \mu_2|\).
we can calculate the separation $S$. For a two-class problem, this is simply

$$ S = \left| \frac{\mu_1 - \mu_2}{\sigma_1 - \sigma_2} \right|, $$

but the difficulties is calculating this for multiple classes is obvious. In Rao's original extension to more that two classes [50], the separation was defined as

$$ S = \frac{\Sigma_b}{\Sigma}. $$

Here, each class had the same covariance $\Sigma$ and the interclass variability is described in terms of the sample covariance of the class means

$$ \Sigma_b = \frac{1}{C} \sum_k (\mu_k - \mu)(\mu_k - \mu)^T, $$

where $\mu$ is the mean of the class means and there are a total of $C$ classes. However, the covariance is rarely known, even though it can be estimated. Alternative approaches include 'one against all', where one class is compared to all other classes, and 'pairwise', which reduces the problem to a series of two-class problems.

Since our example dataset consists of a relatively small number of single-valued measurements, no further data processing or dimension reduction is necessary in this case. This will not be the case for our applications where the data consists of measured continuous time-domain signals. Methods for feature extraction of these signals, and our particular LDA-based method of feature selection, are discussed in Chapter 6. A comparison of methods for feature selection of high-dimensional datasets is given by Hua et al. [51].
3.3 Classification

The goal of supervised statistical pattern recognition is to assign a known feature vector $\mathbf{x}$ to known class $\omega_k$ if the probability that the feature vector belongs to that class is greater than the probability it belongs to any other class. Mathematically stated,

$$P(\omega_k|x) > P(\omega_j|x) \quad j = 1, 2, \ldots, C \quad j \neq k. \quad (3.6)$$

Using the Bayesian formulation [52] we can express the probability that our feature vector belongs to a specific class as

$$P(\omega_k|x) = \frac{p(x|\omega_k)P(\omega_k)}{\sum_{k=1}^{C} p(x|\omega_k)P(\omega_k)}, \quad (3.7)$$

where $P(\omega_k)$ is the a priori probability (the prior probability) and $p(x|\omega_k)$ is the class conditional density. The prior probability is essentially the probability of each class occurring and must sum to unity for $C$ classes ($\sum_{k=1}^{C} P(\omega_k) = 1$). The class conditional density $p(x|\omega_k)$ is the probability of feature vector $\mathbf{x}$ given a class and comes from either a known theoretical or experimental distribution of features.

We can rewrite our decision rule from (3.6) in terms of these probabilities as

$$p(x|\omega_k)P(\omega_k) > p(x|\omega_j)P(\omega_j) \quad j = 1, 2, \ldots, C \quad j \neq k, \quad (3.8)$$

so as long as these terms are known, we can completely calculate the posterior probability $P(\omega_k|x)$ and make the class assignment.

For ease of discussion, we can generalize (3.6) and (3.8) as

$$g_k(\mathbf{x}) > g_j(\mathbf{x}) \quad j = 1, 2, \ldots, C \quad j \neq k, \quad (3.9)$$
where we call \( g_k(x) \) the discriminant function. In (3.6), \( g_k(x) = P(\omega_k|x) \), while in (3.8), \( g_k(x) = p(x|\omega_k)P(\omega_k) \).

For supervised machine learning the class distribution (prior probability distribution \( P(\omega_k) \)) is known, so only the class conditional density \( p(x|\omega_k) \) needs to be estimated. We can categorize the various classifiers we use based on how they attempt this: parametric classifiers assume a form for the class conditional density, while nonparametric classifiers either estimate this density based on patterns in the data or bypass the entire process and estimate the posterior probability directly.

### 3.3.1 Parametric classifiers

One of the most intuitive ways to create a parametric classifier is to assume a normal (Gaussian) form of the class conditional density. This changes the problem of finding unknown \( p(x|\omega_k) \) into finding values appropriate values for the mean \( \mu_k \) and covariance \( \Sigma_k \). This replacement leads to a discriminant function [33]

\[
g_k(x) = \log(P(\omega_k)) - \frac{1}{2} \log(|\Sigma_k|) - \frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k). \tag{3.10}
\]

The problem still remains of creating good estimates for the mean and covariance. These can be estimated from properties of the training set, in particular the maximum likelihood estimate of the mean of the features \( m \) and the sample covariance matrix \( \hat{\Sigma} \)

\[
m = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{3.11}
\]
\[
\hat{\Sigma} = \frac{1}{N} \sum_{i=1}^{N} (x_i - m)(x_i - m)^T. \tag{3.12}
\]

51
Making these replacements in (3.10) \( \left( \mu_k \to m \text{ and } \Sigma_k \to \hat{\Sigma} \right) \) leads to the quadratic discriminant classifier (QDC)

\[
g_k(x) = \log(P(\omega_k)) - \frac{1}{2} \log(|\hat{\Sigma}_k|) - \frac{1}{2}(x - m_k)^T \hat{\Sigma}_k^{-1}(x - m_k) \quad \text{(QDC)}.
\] (3.13)

The linear discriminant classifier assumes all the class covariance matrices \( \Sigma_k \) are the same and can be replaced with a common group covariance matrix \( S_w \), defined as

\[
S_w = \frac{N}{N - C} \sum_{i=1}^{C} \frac{N_i}{N} \hat{\Sigma}_i.
\] (3.14)

This allows a simplified form of the discriminant:

\[
g_k(x) = \log(P(\omega_k)) - \frac{1}{2} m_k^T S_w^{-1} m_k + x^T S_w^{-1} m_k \quad \text{(LDC)}.
\] (3.15)

Two variants of the LDC exist, running the same LDC algorithm on differently preprocessed data. For the KLLDC classifier, the data is projected on the first eigenvectors of the averaged covariance matrix of the classes using the Karhunen-Loeve expansion [53], while the PCLDC classifier projects the data on the first \( N \) eigenvectors of the total dataset using Principal Component Analysis.

For the special case in which \( S_w \) is the identity matrix and the prior probabilities \( P(\omega_k) \) are equal for all \( C \) classes, the LDC discriminant becomes

\[
g_k(x) = -2x^T m_k + m_k^T m_k \quad \text{(NMC)},
\] (3.16)

and is known as the near mean classifier. This corresponds to a measure of the Euclidean distance \( ||x - \mu_k|| \) between the feature vector \( x \) and all \( C \) mean vectors \( \mu_k \). This plain near mean classifier does not assume a normal distribution, but a scaled
version exists that is sensitive to class priors.

The other parametric classifiers that will be used in our analysis are the logistic linear classifier and the Fisher classifier. The logistic linear classifier maximizes the likelihood criterion using the logistic function of shape \( f(x) = \frac{1}{1 + e^{-x}} \), but doesn't scale well to large feature vectors \( \mathcal{O}(1000) \) [54, 55]. The Fisher linear classifier finds the linear discriminant function between the classes by minimizing the errors in the least square sense [56]. Originally a binary classifier, analysis can be expanded to more than two classes using a one-against-all method. For two-class problems, or multi-class problems with equal class prior probabilities, it is equivalent to LDC.

The decision boundaries created by several of these parametric classifiers are shown in Figure 3.4 for a two-dimensional subset of our example glass dataset. Because the headlamp and tableware classes overlap quite severely in the 2-dimensional subset of feature space that we have chosen, Figure 3.5 shows the decision boundaries for the simpler problem of distinguishing between two of the more separated classes in this feature space.

3.3.2 Nonparametric classifiers

Parametric classifiers assume a unimodal density, but most practical problems involve multimodal probability densities. In addition, parametric classifiers assume that the class conditional density follows a known form (typically Gaussian), which is not probable for most real-world datasets. Instead we can use nonparametric classifiers, which estimate the class conditional density directly from the dataset, rather than making these simplified assumptions.

The most common method of estimating the density is to divide the input space
Figure 3.4: The decision boundaries from 4 parametric classifiers are shown on a two-dimensional subset of our glass dataset comparing weight percent sodium (Na) and refractive index (n). In this feature space, the headlamp and tableware classes overlap and are nearly impossible for the classifier to distinguish. Overall classification performance is much better than these boundaries would suggest because we have the benefit of higher dimensions separating the classes more.
Figure 3.5: The decision boundaries from 4 parametric classifiers are shown on a two-dimensional subset of our glass dataset comparing weight percent sodium (Na) and refractive index (n). We have simplified the problem to a 2-class classification in order to highlight differences in the decision boundaries.
(the feature vector \( \mathbf{x} \) of length \( N \)) into equally sized bins \( h \) [34]

\[
\hat{p}(\mathbf{x}) = \frac{\text{\# of } x_i \text{ in same bin as } \mathbf{x}}{Nh}. \tag{3.17}
\]

For a two-dimensional space, this is a histogram and the bins are intervals, while in a three-dimensional space the bins are volumes. For our multidimensional feature space, the bins are hyper-spheres.

If we take the first \( k \) samples that fall into a given interval \( (k \in \mathbb{Z}) \), the density estimate is

\[
\hat{p}(\mathbf{x}) = \frac{k}{NV}, \tag{3.18}
\]

where we have changed the bin symbol \( h \) to \( V \) to emphasize that the bins are a hyper-volume in our multidimensional feature space. This allows us to write the class conditional and prior probabilities as (changing our index to \( j \) for this time only to avoid confusion)

\[
p(\mathbf{x} | \omega_j) = \frac{k_j}{N_jV} \tag{3.19}
\]

\[
P(\omega_j) = \frac{N_j}{N} \tag{3.20}
\]

and to write the discriminant as

\[
g_j(\mathbf{x}) = \frac{k_j}{N_jV} \frac{N_j}{N} \text{ (KNN).} \tag{3.21}
\]

This is the k-nearest neighbor classifier, and assigns \( \mathbf{x} \) to the most frequently populated class in the volume. This is a popular nonparametric classifier, but is prone to overtraining [57, 58].
The Parzen classifier generalizes this approach, with a density estimate

\[
\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{V} \phi \left( \frac{x - x_i}{h_n} \right),
\]

(3.22)

where \( h_n \) is the length of one edge of the \( d \)-dimensional volume such that \( V = h_n^d \). The windowing function \( \phi \left( \frac{x - x_i}{h_n} \right) \) is equal to unity if \( x_i \) falls within a hypercube of volume \( V \) and is zero otherwise, essentially interpolating so that each \( x_i \) contributes based on its distance from \( x \).

The final nonparametric classifier we will consider is the Support Vector Machine (SVM), also referred to as the Support Vector Classifier (SVC) [59, 60]. This is a binary classifier that maps feature vectors to an even higher dimensional space to find a hyperplane that maximizes the margin between classes. The discriminant function is

\[
g(x) = w^T x + w_0 \quad \text{(SVC),}
\]

(3.23)

and since this is a binary classification \( g(x) > 0 \) assigns \( x \) to class \( \omega_1 \), while \( g(x) < 0 \) assigns \( x \) to class \( \omega_2 \) for some set of weights \( w \). A kernel formulation of this classifier is often used to speed up the computation [61].

Since the SVC is a binary classifier, we need a method to expand this analysis to multi-class problems. There are two main methods to accomplish this: ‘one-against-one’, where the binary classification is done for all possible pairs of classes and ‘one-against-all’, where each original class is compared to a new (temporary) class comprised of data from other classes.

One popular nonparametric classifier, neural networks, is noticeably absent from our discussions. Neural networks, and other related genetic algorithms, tend to have a ‘black box’ nature that hinders utilization of known information about the physical problem to improve classification performance. Due to this, as well as the large
amount of training time for neural networks, we will not be including these classifiers in our analysis.

The decision boundaries created by a several of these nonparametric classifiers are shown in Figure 3.6 for the same two-dimensional subset of our example glass dataset. As for the parametric classifiers, decision boundaries for a more separable 2-class problem is shown in Figure 3.7

3.3.3 Improving classifier performance

For our classification tests we are using the openly-distributed PRTools toolbox created at the Delft University of Technology [62]. This provides a standardized, easy-to-use set of Matlab programs for analysis. Our interest in pattern classification is application of existing algorithms to new and interesting problems, rather than creating new classifiers. Because we have constructed a small-dimensional feature vectors, we can easily test a number of parameters of the classification, as well as a large number of classifiers.

A number of classifiers are provided with PRTools, and those used in our analysis are listed in Table 3.3. Many other classifiers, variants of basic classifiers, and methods to combine classifiers exist and in general allow slight improvements in classification performance. Instead of focusing on these nuances, our approach is to improve performance by creating a good feature vector representation of our data as input.

Before we attempt classification we must separate our data into training and testing (or validation) datasets. By withholding a subset of the data for testing the classifier's performance we can eliminate artificial improvements to the classification performance that come from using training data for testing. Classification performance over a number of train/test ratios will also give some indication of the robustness of that particular classifier, and help prevent overtraining. Table 3.4 shows the overall
Figure 3.6: The decision boundaries from 4 nonparametric classifiers are shown on a two-dimensional subset of our glass dataset comparing weight percent sodium (Na) and refractive index (n). In this feature space, the headlamp and tableware classes overlap and are nearly impossible for the classifier to distinguish. Overall classification performance is much better than these boundaries would suggest because we have the benefit of higher dimensions separating the classes more.
Figure 3.7: The decision boundaries from 4 nonparametric classifiers are shown on a two-dimensional subset of our glass dataset comparing weight percent sodium (Na) and refractive index ($n$). We have simplified the problem to a 2-class classification in order to highlight differences in the decision boundaries.
Table 3.3: A number of parametric and non-parametric classifiers included in PRTools are used in our analysis.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parametric classifiers</strong></td>
<td></td>
</tr>
<tr>
<td>qdc</td>
<td>Quadratic classifier assuming normal densities</td>
</tr>
<tr>
<td>udc</td>
<td>Quadratic classifier assuming normal uncorrelated densities</td>
</tr>
<tr>
<td>ldc</td>
<td>Linear classifier assuming normal densities</td>
</tr>
<tr>
<td>kldc</td>
<td>Linear classifier by KL expansion of common cov matrix</td>
</tr>
<tr>
<td>pcldc</td>
<td>Linear classifier by PCA expansion on the joint data</td>
</tr>
<tr>
<td>nmc</td>
<td>Nearest mean classifier</td>
</tr>
<tr>
<td>nmsc</td>
<td>Scaled nearest mean classifier</td>
</tr>
<tr>
<td>loglc</td>
<td>Logistic linear classifier</td>
</tr>
<tr>
<td>fisherc</td>
<td>Minimum least square linear classifier</td>
</tr>
<tr>
<td><strong>Nonparametric classifiers</strong></td>
<td></td>
</tr>
<tr>
<td>knnc</td>
<td>k-nearest neighbor classifier</td>
</tr>
<tr>
<td>parzenc</td>
<td>Parzen classifier</td>
</tr>
<tr>
<td>parzendc</td>
<td>Parzen density based classifier</td>
</tr>
<tr>
<td>kernelc</td>
<td>General kernel/dissimilarity based classification</td>
</tr>
<tr>
<td>perlc</td>
<td>Linear perceptron</td>
</tr>
<tr>
<td>svc</td>
<td>Support vector classifier</td>
</tr>
<tr>
<td>nusvc</td>
<td>Support vector classifier</td>
</tr>
<tr>
<td>rbsvc</td>
<td>Radial basis SV classifier</td>
</tr>
<tr>
<td>treec</td>
<td>Construct binary decision tree classifier</td>
</tr>
</tbody>
</table>

classification performance (mean overall correct percent) for the classification of non-window glasses for several different classifiers.

From Table 3.4 we see that each classifier responds to different train/test ratios in a different manner. For some, like QDC, classification performance increases with an increased amount of training data but require a minimum ratio (here, 0.6) to attain a reasonable performance level. Other classifiers are less dependent on the train/test ratio, but performance still increases slightly as the amount of training data increases.
Table 3.4: The mean overall correct percent (after 20 runs) of the 3-class, non-window classification task shows the impact of the train/test ratio for several common parametric and nonparametric classifiers. For clarity, the variance of the mean overall correct percent is not shown, but tends to be < 0.1 for all train/test ratios.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>0.25</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>qdc</td>
<td>0.18</td>
<td>0.37</td>
<td>0.78</td>
<td>0.81</td>
<td>0.89</td>
</tr>
<tr>
<td>ldc</td>
<td>0.77</td>
<td>0.82</td>
<td>0.84</td>
<td>0.85</td>
<td>0.87</td>
</tr>
<tr>
<td>udc</td>
<td>0.26</td>
<td>0.19</td>
<td>0.15</td>
<td>0.15</td>
<td>0.13</td>
</tr>
<tr>
<td>nmsc</td>
<td>0.79</td>
<td>0.83</td>
<td>0.87</td>
<td>0.87</td>
<td>0.93</td>
</tr>
<tr>
<td>knnc</td>
<td>0.74</td>
<td>0.81</td>
<td>0.82</td>
<td>0.79</td>
<td>0.86</td>
</tr>
<tr>
<td>parzenc</td>
<td>0.76</td>
<td>0.78</td>
<td>0.77</td>
<td>0.78</td>
<td>0.82</td>
</tr>
<tr>
<td>nusvc</td>
<td>0.79</td>
<td>0.84</td>
<td>0.87</td>
<td>0.90</td>
<td>0.89</td>
</tr>
<tr>
<td>treec</td>
<td>0.72</td>
<td>0.78</td>
<td>0.81</td>
<td>0.81</td>
<td>0.85</td>
</tr>
</tbody>
</table>

What we don’t have here is an example of an overtrained classifier, where performance would actually decrease after a certain train/test ratio.

Our analysis shows that a train/test ratio of 0.6 or greater will provide an acceptable level of performance for comparisons without overtraining. This also allows a testing (validation) set that consists of more than a single instance.

To get a better idea of overall classification performance and ensure that individual observations aren’t overly helping (or hurting) the classification performance, the data used for testing and training is randomized for each run. This gives us an average classification performance and allows us to use standard deviation of correct classifications as a measure of classification repeatability, as well as to test the convergence of different classifiers, an example of which is shown in Figure 3.8.

Finally, it is important to note that we shouldn’t exclude any particular classifiers from our analysis based on previous knowledge. The ‘No Free Lunch’ theorem says that in the absence of a priori information about our problem there is no reason to
Figure 3.8: As the number of runs increase, the classification performance stabilizes. (Non-window glass classification, 0.7 train, knnc classifier)

prefer one classifier over another [63]. The consequence of this is that if one classifier seems to provide better results, it is simply because it is a better fit for the current data, rather than being an overall better classifier.

This means that in order to ensure we are getting the best possible performance we must run the classification routine for all possible classifiers. For example, even though we don’t expect our data to have underlying probability densities, we shouldn’t exclude the parametric classifiers based on that assumption. Once again, this type of analysis is practical because we have created an small, information-dense feature vector that efficiently represents our original data.
3.4 Visualizing classification results

The most popular representation of classification performance is the confusion matrix (also called a contingency table), whose rows contain the number of instances of each label and whose columns show the number of times an instance has been classified as that label.

For a simple binary classification, the confusion matrix is a simple $2 \times 2$ matrix,

\[
\begin{array}{c|cc}
\omega_1^a & a & b \\
\omega_2^a & c & d \\
\hline
\omega_1^p & \omega_2^p \\
\end{array}
\]  \hspace{1cm} (3.24)

where the label $\omega_1^a$ is the actual class and $\omega_2^p$ is the predicted class. For this simplified case, we can define a number of useful metrics that tell us about the classification performance (Table 3.5) [64].

Table 3.5: Metrics defined using the $2 \times 2$ confusion matrix from a binary classification can be extended to a more general $n \times n$ multi-class confusion matrix.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>$\frac{a+d}{a+b+c+d}$</td>
</tr>
<tr>
<td>Sensitivity (True positive rate)</td>
<td>$\frac{d}{c+d}$</td>
</tr>
<tr>
<td>Specificity (True negative rate)</td>
<td>$\frac{a}{a+b}$</td>
</tr>
<tr>
<td>Precision</td>
<td>$\frac{d}{b+d}$</td>
</tr>
<tr>
<td>False positive rate</td>
<td>$\frac{b}{a+b}$</td>
</tr>
<tr>
<td>False negative rate</td>
<td>$\frac{c}{c+d}$</td>
</tr>
</tbody>
</table>

These terms are slightly more intuitive if class 1 $\omega_1$ is thought of as 'negative' and class 2 $\omega_2$ is thought of as 'positive'. Then, $a$ is the number of correct predictions that an instance is negative and $b$ is the number of incorrect predictions that an instance
is positive. Likewise, $c$ is the number of incorrect predictions that an instance in negative and $d$ is the number of correct predictions that an instance is positive.\footnote{Going one step deeper, when the classifier makes the decision between positive and negative for a given instance, it is using some threshold $h$ as a comparison, which means that all of these metrics are further a function of $h$.}

In particular, sensitivity ($\chi$) and specificity ($\psi$) will turn out to be quite useful. High sensitivity values indicate good classification when the predicted class is the same as the actual class, while high specificity values indicate good non-classification of classes that are not the actual class.

For our purposes we will show a normalized confusion matrix, where the value in each cell is the percent correct classification rather than the number of instances. Using this method, the values of each row should sum to unity and a perfect classification would have values of 1 on the main diagonal and zero everywhere else. We will also view the confusion matrix as a color-mapped image to facilitate quick analysis (Figure 3.9). This arrangement makes it easy to determine not only the accuracy of the classification, but how the classification is mis-labeling instances. For example, a classification that is 'close' (with the majority of incorrectly labeled instances falling in nearby classes) is immediately distinguishable from a classification where a majority of the mislabeled instances were from distant classes.

While the confusion matrix provides much useful information about the classification, it is less than optimal for comparing performance between classifiers due to its lack of a single-value metric. Since the 'No Free Lunch' theorem requires us to run the classification routine using many classifiers and parameters of the classification, we require a consistent method to evaluate the classification results.

The most obvious such method is to develop our own metric from the confusion matrix. Any such metric will cause us to lose information about the details of the classification, but this is an acceptable trade-off in order to reduce the number of
Figure 3.9: The average confusion matrix for the task of categorizing non-window glass into one of three known classes shows us details of the classification performance. The header tells us the classifier used, the number of times the classification routine ran (with randomized train/test data each time), the number of instances present in the dataset, the percent of the dataset used as training data, and the number of instances of each class present in the training and testing data. The header also shows our main performance metric, the mean overall percent correct, along with a standard deviation measure of variance. The mean correct per class is the value of the diagonal in the colormapped image, included for accuracy. The confusion matrix itself compares the estimated class labels to the known true classes.
possible classifiers. When a small handful of 'good' classifiers have been identified for
a specific problem, more detailed investigation of the classification performance will
once again return to analysis of the confusion matrix.

We have chosen the mean overall percent correct as our metric to compare clas-
sifiers. This is simply calculated by taking the average of the overall percent correct
over all of the classification runs, typically $\mathcal{O}(10 - 20)$. The standard deviation of
this overall percent correct can be calculated in a similar manner to give some sense
of the variation in classification runs.

Provost et al. point out a number of issues with using classification accuracy to
compare classifiers, in particular the assumptions of a known class distribution and
equal misclassification costs [65]. The former is not an issue for supervised learning,
but the treating false positives and false negatives the same may be problematic
for certain applications. One such example is the classification of flaw depths in a
material, where misclassification of a 1% flawed sample as 2% flawed is much less
problematic than misclassification of the same sample as 50% flawed [18]. However,
we consider this assumption of equal misclassification costs acceptable for both our
example problem of differentiating types of glass and our real problem of acoustic
vehicle classification discussed in Chapter 6, and will use mean overall percent correct
as a metric to compare classifier performance.

One alternative method of comparing classifier performance is the analysis of
Remote Operating Characteristic (ROC) curves [66]. These curves show how the
sensitivity and specificity change as a function of the threshold $h$, and are created by
plotting $\chi(h)$ versus $1 - \psi(h)$ (Figure 3.10).

On such plots, the diagonal line $y = x$ indicates random guessing, so any classifier
with a ROC curve above this line (in the upper left-hand corner of the plot) are
performing better than random guessing by incorporating additional information

67
Figure 3.10: The ROC curve for the binary classification of a glass sample as coming from a window or from a non-window has a high AUC value, indicating good performance. This is confirmed by a mean overall correct classification rate of 93%. The random guessing line of $y = x$ is indicated by the light blue dashed line about the problem. Since the lower right-hand corner (below the diagonal line) indicates performance worse than random guessing, it should be empty. Perfect classification performance occurs at the $x, y$ point $(1, 1)$, corresponding to the point $(0, 1)$ on the ROC plane.

Once again, we require a single-value metric to most easily compare classifiers. The area under the ROC curve (AUC) is a commonly used metric. Higher AUC values tend to correspond to better classification performance, though this is not a

---

$^5$If a ROC curve is below the random guessing line, it means that the classifier has additional information about the problem and is using it exactly wrong.
strict rule [67, 68].

Extensions of ROC analysis to multi-class classification problems is possible, but recent work has led to questions about the usefulness of AUC measurements [69]. Since the objections to accuracy-based comparisons don’t apply to our datasets, we will use simpler accuracy measurements obtained directly from the confusion matrix to compare performance between classifiers.

3.5 Some results

Finally, we will discuss a few results from classification tests run on our glass dataset to illustrate some general principles about pattern classification.

As discussed above, based on physical knowledge of the glass dataset we have several classification tasks to perform. These are:

1. Non-window type: Categorizing the non-window glass samples into one of three known classes (container, tableware, headlamps)

2. Window vs. non-window: A binary classification of all instances present in the dataset as either window glass or non-window glass

3. Overall type: The most specific task, which attempts to classify each instance in the dataset into one of the six known classes (building float-processed, building non-float processed, vehicle, container, tableware, headlamps)

The average classification performance (using mean overall correct percent) for each of these classification tasks is given in Table 3.6. Nine parametric and nine nonparametric classifiers were used to generate the average confusion matrices, and specific results for each classifier are presented in Table 3.7.
Table 3.6: A comparison of the mean overall correct percent (20 runs, 0.6 train/test ratio), averaged over 8 parametric and 9 nonparametric classifiers for three different classification tasks from our glass dataset. These tasks are differentiate origin of non-window, determine if sample comes from window glass or non-window glass, and classify the glass as one of six possible types. The UDC classifier was a clear outlier and was removed from this analysis (included in window vs. non-window).

<table>
<thead>
<tr>
<th>Classifier group</th>
<th>Non-window type</th>
<th>Window vs. non-window</th>
<th>Overall type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parametric</td>
<td>0.82</td>
<td>0.92</td>
<td>0.58</td>
</tr>
<tr>
<td>Nonparametric</td>
<td>0.83</td>
<td>0.93</td>
<td>0.60</td>
</tr>
<tr>
<td>Overall average</td>
<td>0.83</td>
<td>0.92</td>
<td>0.59</td>
</tr>
</tbody>
</table>

In general, we have good performance, and once the outlying UDC classification is removed, any difference in performance between parametric and nonparametric classifiers disappears. As a comparison, in the case of the three class non-window classification, random guessing corresponds to a classifier performance of 33%. Likewise, 50% performance is random guessing for the binary window vs. non-window classification, and 20% is random guessing for the difficult task of classifying a sample as one of the 5 known types of glass.

As would be expected, the best performance comes from the simplest classification task, the binary window vs. non-window classification, with an average mean overall correct rate of 92%. The next best classification task was for the three-class non-window data, with an average mean overall correct rate of 83%. In addition to the lower overall performance, the performance of individual classifiers is more varied.

The worst performing task was the classification into the six classes. The average mean overall correct rate was only 59%, which is still much better than random guessing. Classification performance in this range would be an excellent candidate for some of the classifier combination techniques meant to improve accuracy of low-scoring classifiers.
Table 3.7: The mean overall correct percent (20 runs, 0.6 train/test ratio) is a measure of classifier performance for three different classification tasks from our glass dataset. These tasks are differentiate origin of non-window, determine if sample comes from window glass or non-window glass, and classify the glass as one of 5 possible types.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Non-window type</th>
<th>Window vs. non-window</th>
<th>Overall type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parametric classifiers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qdc</td>
<td>0.74 ± 0.10</td>
<td>0.93 ± 0.02</td>
<td>0.55 ± 0.06</td>
</tr>
<tr>
<td>udc</td>
<td>0.15 ± 0.02</td>
<td>0.90 ± 0.02</td>
<td>0.04 ± 0.01</td>
</tr>
<tr>
<td>ldc</td>
<td>0.83 ± 0.11</td>
<td>0.92 ± 0.03</td>
<td>0.62 ± 0.04</td>
</tr>
<tr>
<td>klldc</td>
<td>0.84 ± 0.06</td>
<td>0.92 ± 0.02</td>
<td>0.61 ± 0.06</td>
</tr>
<tr>
<td>pcldc</td>
<td>0.82 ± 0.11</td>
<td>0.92 ± 0.03</td>
<td>0.62 ± 0.04</td>
</tr>
<tr>
<td>nmc</td>
<td>0.83 ± 0.10</td>
<td>0.91 ± 0.02</td>
<td>0.48 ± 0.04</td>
</tr>
<tr>
<td>nmsc</td>
<td>0.85 ± 0.05</td>
<td>0.92 ± 0.02</td>
<td>0.55 ± 0.04</td>
</tr>
<tr>
<td>loglc</td>
<td>0.85 ± 0.07</td>
<td>0.91 ± 0.03</td>
<td>0.63 ± 0.05</td>
</tr>
<tr>
<td>fisherc</td>
<td>0.83 ± 0.09</td>
<td>0.92 ± 0.03</td>
<td>0.59 ± 0.04</td>
</tr>
<tr>
<td><strong>Nonparametric classifiers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>knnc</td>
<td>0.84 ± 0.08</td>
<td>0.94 ± 0.02</td>
<td>0.64 ± 0.06</td>
</tr>
<tr>
<td>parzenc</td>
<td>0.78 ± 0.09</td>
<td>0.93 ± 0.03</td>
<td>0.58 ± 0.05</td>
</tr>
<tr>
<td>parzendc</td>
<td>0.73 ± 0.07</td>
<td>0.93 ± 0.03</td>
<td>0.56 ± 0.02</td>
</tr>
<tr>
<td>kernelc</td>
<td>0.83 ± 0.07</td>
<td>0.93 ± 0.03</td>
<td>0.66 ± 0.05</td>
</tr>
<tr>
<td>perlc</td>
<td>0.87 ± 0.09</td>
<td>0.92 ± 0.02</td>
<td>0.51 ± 0.07</td>
</tr>
<tr>
<td>svc</td>
<td>0.88 ± 0.06</td>
<td>0.92 ± 0.02</td>
<td>0.58 ± 0.03</td>
</tr>
<tr>
<td>nusvc</td>
<td>0.86 ± 0.09</td>
<td>0.92 ± 0.02</td>
<td>0.62 ± 0.05</td>
</tr>
<tr>
<td>rbsvc</td>
<td>0.84 ± 0.01</td>
<td>0.94 ± 0.02</td>
<td>0.60 ± 0.01</td>
</tr>
<tr>
<td>treec</td>
<td>0.82 ± 0.02</td>
<td>0.92 ± 0.04</td>
<td>0.63 ± 0.06</td>
</tr>
</tbody>
</table>
Chapter 4

rMary - a walking-speed mobile sensor platform

Creating an autonomous robot able to act independently of human control has long been an area of active research in robotics. New low-cost sensors and recent advances in signal processing necessary to analyze large amounts of streaming data has only increased the number of researchers focusing on autonomous robotics, buoyed by a greater public awareness of the field.

In particular, DARPA-funded competitions have enabled focused efforts to create autonomous vehicles and humanoid robots. The 2004 DARPA Grand Challenge and follow-on 2007 DARPA Urban Challenge [70] focused on creating autonomous vehicles that could safely operate over complex courses in rural and urban environments. These competitions rapidly expanded the boundaries of the field, leading to recent near-commercial possibilities such as Google’s self-driving cars [71, 72]. Similarly, the recent and rapid rise of Unmanned Aerial Vehicles (AUVs) has led to a large amount of research in designing truly autonomous drone aircraft. Designing autonomous vehicles, whether surface and aerial, comes with its own difficulties, namely collecting
and interpreting data at a fast enough rate to make decisions. This requires expensive sensors that only large research programs can afford. Commercialization of such technologies will require lower-cost alternative sensor modalities.

Likewise, interest in creating a humanoid robot has received a boost from the currently-running DARPA Robotics Challenge [73]. This challenge focuses on the ability of existing tethered robots to perform complex tasks such as locomotion over rough surfaces, vehicle entry and driving, and operation of human power tools. If the previous DARPA Grand challenge for autonomous vehicles is any indication, this challenge will go a long way to advance the field, but serious commercialization and wide adoption of the technology seems to be many years away.

A more tractable problem is the design of walking-speed robotic platforms. Compared to surface or air vehicles, the lower-speed of such platforms allows the use of low-cost sensor modalities that take longer to acquire and analyze data, since more time is available to make a decision. Using commercially available wheeled platforms (such as all-terrain vehicles) shifts focus from the engineering problems in creating a humanoid robot to the types of sensors used and how such data can be combined. For these reasons, we will focus on the analysis of different sensor modalities for a walking speed robot. This chapter will give a brief background on existing commercial technologies before discussing the benefits of different sensors using data collected from our current robotic platform, rMary.

\footnote{The robotic platforms are tethered because current battery capabilities are not able to meet the power requirements.}
4.1 Towards an autonomous walking-speed robotic platform

The goal in autonomous robotics is to create a robot with the ability to perform tasks normally accomplished by a human. An added bonus is the ability to do tasks that are too dangerous for humans, such as entering dangerous environments in disaster situations. A more mundane example is having a robotic assistant fetch a cup of coffee from across the street. While this is a simple task from a human point-of-view, successful completion requires the robot to exhibit many human features. Just a few of the requirements involved are sensing (where is the door?), navigation (how to get there?), mechanical (how to open the door?), and human interaction (what to do if encountering someone else on the way?). Beyond a thought experiment, an example of giving an existing bipedal robot such a task was presented at the 2013 IEEE International Conference on Robotics and Automation [74]. In particular this task required navigating using multiple (provided) maps, operating an elevator (though refusing to board if humans were present for safety reasons), opening heavy, transparent glass doors, and accepting objects passed from humans. Even though this is only a subset of the necessary tasks, it highlights the key issues that must be solved to design a useful autonomous robot: mechanical control, human interaction, navigation, and sensing.

We will not discuss the mechanical and human interaction aspects of autonomous robotics any further. Likewise, while we are not concerned with programming a robot to navigate through a set course, some aspects of navigation are important as they relate to the ability of a robot to sense its environment. A good example comes from iRobot's Roomba, a small saucer-shaped disc that rolls around a house and vacuums the floors. Since the Roomba's primary sensor is a touch-based switch, it
only knows about a colliding with an object after the fact. Without knowledge of the room layout or location of obstacle (i.e., a map) it cannot determine the most efficient way to accomplish its task, instead depending on a pseudo-random method in an attempt to cover the entire floor. At the other end of the spectrum are SLAM-capable (Simultaneous Location And Mapping) robots that create a map as they move through a space. These platforms tend to require large, expensive sensors such as LIDAR (Lig ht Detection and Ranging) that are not yet feasible for many commercial applications. This is especially true in unstructured environments where few assumptions can be made about objects that will be encountered and navigation is more difficult.

Regardless of the environment or manner in which the robot navigates said environment, the key to allowing a robot to make sound decisions is its ability to learn about the environment. This issue of sensing is our main focus.

4.1.1 Sensor modalities for mobile robots

We can classify sensor modalities as 'active' or 'passive' depending on whether they transmit a signal (i.e., radar) or use information already present in the environment (i.e., an RGB image), respectively. The use of passive sensors is often preferred to to reduce the possibility of detection in covert operations and reduce annoyance.

Another important consideration is the range at which a given sensor performs. For example, imaging systems can provide detailed information about objects near the sensor but may not detect fast-moving hazards (such as an oncoming vehicle) at a great enough distance to allow a robot time to evade. Long-range sensors such as radar or LIDAR are able to detect objects at a greater distance, giving the robot more time to maneuver out of the way. This long-range detection often requires expensive sensors which don't provide detailed information about the target. A combination
of near- and long-range sensors will give a robot the most information about its environment.

Once the sensor has measured some information about its environment, the robot needs to know how to interact. A real-world example of this difficulty comes from agriculture, where smart machines have the ability to replace human workers in repetitive tasks. One agricultural application in particular is the thinning of lettuce, where human laborers paid by the acre thin healthy plants unnecessarily. A new, robotic 'Lettuce Bot' is towed behind a tractor, imaging individual lettuce plants as it passes and using computer vision algorithms to comparing these images to a database of over a million images to decide which plants to remove by dousing them with a concentrated dose of fertilizer [75]. Though this machine claims 98% accuracy while driving at 2 kph and may be cost-competitive with manual labor, it also highlights issues with image-based analysis on mobile robots. Creating a large enough database for different types of lettuce is a monumental task, given the different colors, shapes, soil types, and other variables. Even the sun creates problems, causing shadows that are difficult for the computer vision software to correctly match. Shielding the sensor and restraining the image database to a particular geographical region (thereby reducing the number of lettuce variants and soil types) allows these techniques to work for this particular application but the approach is not scalable to more unstructured environments. While the human brain has evolved to process images quickly and easily, automated interpretation of images is a difficult problem. Using non-imaging sensors can ease the signal processing requirements, but requires sophisticated machine learning techniques to deal with large amounts of abstract data.

In addition to the range limitations of different sensors and the difficulty in analyzing the resulting data, individual sensor modalities tend to work better in particular environmental conditions. For example, a webcam can create detailed images in
a well-lit environment but fail to provide much useful information on a dark night while passive infrared images can detect subtle changes in emissivity from surfaces in a variety of weather conditions. Because of the limitations of individual sensors, intelligent combinations of complementary sensors must be used to create the most robust awareness of an unstructured environment. The exact manner in which these modalities are combined is referred to as data fusion [76].

Our focus is the performance of different sensor modalities in real-world, unstructured environments under a variety of environmental conditions. Our robotic sensor platform, rMary, contains a number of both passive and active sensors. Passive vision-based sensors include a standard RGB webcam and infrared sensors operating in both the near-infrared and long-wave regions of the infrared spectrum. Active sensors include a three-dimensional depth mapping system that uses infrared projection, a simple radar system, and a speaker/microphone combination to perform acoustic echolocation in the audible range. The usefulness of these acoustic echolocation measurements will be further discussed in Chapter 6, where we will apply machine learning algorithms to this data to automatically detect and classify oncoming vehicles at long distances.

4.2 Investigation of sensor modalities using rMary

To collect data in unstructured outdoor environments, we have created a mobile sensor platform with multiple sensors (Figure 4.1). This platform, named rMary, was first placed in service in 2003 to collect infrared data from passive objects [77]. The robot is remotely operated using a modified RC aircraft controller and is steered using four independent drive motors synced to allow agile skid-steering. Power is supplied to these motors from a small battery bank built into the base of the platform, where the
control electronics are also located. The low center of gravity, inflatable rubber tires, and a custom-built suspension system allow off-road transit to acquire measurements in otherwise inaccessible locations.

In 2011, rMary was stripped to the base and fit with a new frame. This creates a platform to mount additional sensors which can be easily modified for future sensor upgrades. The entire frame is attached to the mobile platform with a series of four bolts, allowing easy access to the internals of rMary’s drive system. An inverter draws power from rMary’s built in batteries to provide standard 120V electrical outlets for the parametric array and Microsoft Kinect. There is also enough room to add an additional battery bank to allow for extended sorties, but the weight of the additional lead-acid batteries can bottom out the suspension, making off-road travel difficult.

The current sensors on rMary include:

- Raytheon ControlIR 2000B long-wave infrared camera
- Microsoft Kinect (2010)
  - Active IR projector
  - IR and RGB sensors
  - 4-channel microphone array
- Sennheiser Audiobeam acoustic parametric array
- Coffee can FMCW ISM-band radar

A parabolic dish microphone can also be attached, but the Kinect microphone array provides superior audio recordings. Sensor control and data acquisition is accomplished using a low-powered Asus EeePC 1000h Linux netbook. This underpowered laptop was deliberately used to show that data can be easily acquired with commodity hardware.
Figure 4.1: The lone sensor on the first rMary (2006, left) was a thermal imaging camera housed in the front box to protect it from the elements. An external frame was added to rMary (2011, right) to create mounting locations to add additional sensors such as the Microsoft Kinect and an acoustic parametric array.

The computer's single internal USB hub does restrict the number of simultaneous data streams, which only became an issue when trying to collect video data from multiple hardware devices using non-optimized drivers. Each of the sensors, whose location is shown in Figure 4.2, will be discussed in the sections that follow.

4.2.1 Thermal infrared (IR)

A Raytheon ControlIR 2000B infrared camera couples a long-wave focal plane array microbolometer detector to a 50 mm lens to provide $320 \times 240$ resolution at 30 Hz over a $18^\circ \times 13.5^\circ$ field of view. Although thermal imaging cameras are now low-cost
Figure 4.2: The newest iteration of the rMary sensor platform contains a forward-looking long-wave infrared camera, mounted upright in an enclosure for stability and weather protection, an acoustic parametric array, the Microsoft Kinect sensor bar, and a coffee can radar. All sensors are powered by the on-board battery bank and controlled with a netbook computer running Linux.
and portable enough to be used by home inspectors for energy audits, this was one of the few uncooled, portable infrared imaging systems available when first installed in 2006.

These first experiments showed that the sensor was able to characterize passive (non heat-generating) objects through small changes in their thermal signatures [78, 79]. The sensor measures radiance in the long-wave region (8-15 µm) of the infrared spectrum where radiation from passive objects is maximum (Figure 4.3).

Figure 4.3: The long-wave infrared band (highlighted in blue) yields the highest thermal radiance over a range of passive objects likely to be encountered by an autonomous robot operating on Earth, represented by their theoretical blackbody radiation curves. (Image used with permission from [77])

For stability and protection from the elements, the camera is mounted vertically in an enclosed locker. An polished aluminum plate with a low emissivity value makes
a good reflector of thermal radiation and allows the camera to image objects in front of rMary. Figure 4.4 shows several examples of images of passive objects acquired with the thermal infrared camera, both indoors and outside.

4.2.2 Kinect

Automated interpretation of images from the thermal infrared camera requires segmentation of the images to distinguish areas of interest, which can be a difficult image processing task. In addition, the small field-of-view and low resolution of the infrared camera used here led us to investigate possible alternatives. While there are still relatively few long-wave thermal sensors with enough sensitivity to measure the small differences in emissivity between passive objects, other electronics now contain infrared sensors.

One of the most exciting alternatives was the Microsoft Kinect, released in November 2010 as an accessory to the Xbox 360 gaming console. The Kinect was immensely popular, selling millions of units in the first several months, and integrates active infrared illumination, an IR sensor, and an RGB camera to output 640 × 480 RGB-D (RGB + depth) video at 30 Hz. It also contains a tilt motor, accelerometer, and 4-channel microphone array, all at total cost of less than USD $150 (Figure 4.5).

Access to this low-cost suite of sensors is provided by two different open source driver libraries: libfreenect [80], with a focus on audio support and motor controls and OpenNI [81], with greater focus on skeletal tracking and object segmentation. Other specialized libraries such as nestk [82] used these drivers to provide high-level functions and ease of use. In June 2011, Microsoft released their own SDK that provides access to the raw sensor streams and high-level functions, but these libraries only work in Windows 7 and are closed-source with restrictive licenses [83]. In addition, Microsoft changed the license agreement in March 2012 to require use of the ‘Kinect for Windows’
Figure 4.4: Examples of passive objects imaged with the thermal IR camera include (clockwise from top left) a car in front of a brick wall, a tree trunk with foliage, a table and chairs in front of a bookcase, and a window in a brick wall.
Figure 4.5: The Microsoft Kinect includes a webcam, active IR sensor, microphone array, and more to capture depth-mapped video and multi-channel audio for under $150 USD.

sensor instead of the identical but cheaper Kinect sensor for Xbox.

We investigate the usefulness and limitations of the Kinect sensor for robotics, particularly the raw images recorded from the infrared sensor and the depth-mapped RGB-D images. Since our application is more focused on acquiring raw data for later processing than utilizing the high-level skeletal tracking algorithms, we are using the libfreenect libraries to synchronously capture RGB-D video and multi-channel audio streams from the Kinect.

The Kinect uses a structured light approach similar in principle to [84] to create a depth mapping. An infrared projector emits a known pattern of dots, allowing the calculation of depth based on triangulation of the specific angle between the emitter and receiver, an infrared sensor with $1280 \times 1024$ resolution. The projected pattern is visible in some situations in the raw image from the infrared sensor (to which the open-source drivers allow access). To reduce clutter in the depth mapping, the infrared sensor also has a band-stop filter at the projector's output frequency of 830 nm. The Kinect is able to create these resulting $640 \times 480$ resolution, 11-bit depth-mapped
images at video frame rates (30 Hz). The stated range of the depth sensing is 1.2-3.5 m, but in the right environments can extend to almost 6 m. An example of this image for an indoor environment is shown in Figure 4.6, along with the raw image from the IR sensor and a separate photograph for comparison.

In addition to this colormapped depth image, the depth information can also be overlaid on the RGB image acquired from a 1280 x 1024 RGB sensor to create a three-dimensional point-cloud representation (Figure 4.7).

Since the Kinect was designed to work as an accessory to a gaming system, it works well in indoor environments, and others have evaluated its applications to indoor robotics, object segmentation and tracking, and three-dimensional scanning [85, 86].

Figure 4.8 shows a sampling of the raw IR and depth-mapped images for several outdoor objects. The most visible feature when compared to images acquired in indoor environments is that the raw infrared images are very well illuminated, or even over-exposed. Because of this, the projector pattern is difficult to detect in the infrared image, and the resulting depth-mapped images don’t tend to have much structure. There is more likely to be useful depth information if the object being imaged is not in direct sunlight and/or is located very close to the sensor.

To understand why this behavior occurs, we take a closer look at the infrared spectrum (Figure 4.9). Unlike the thermal IR camera which operates in the long-wave region of the IR regime, the Kinect’s infrared sensor operates in the near-infrared. This is necessary so that a distance can be calculated from the projected image, but the proximity of the near-infrared to the visible spectrum allows the sensor to become saturated (Figure 4.10).

Figure 4.11 shows a series of images of the same scene as the sun emerges from behind a cloud. As there is more sunlight, the infrared sensor becomes saturated and no depth mapping can be constructed.
Figure 4.6: In an indoor environment, the Kinect is able to take a raw infrared image (top) and convert it to a corresponding depth-mapped image (middle), which overlays depth information on an RGB image. The speckle pattern barely visible on the raw infrared image is the projected infrared pattern that allows the Kinect to create this depth mapping. As shown in this mapping, darker colored surfaces, such as the desk chair on the left of the image, are closer to the sensor while lighter colors are farther away. Unexpected infrared reflectors can confuse this mapping and produce erroneous results, such as the light fixture in the center of the image. The bottom image is a photograph of the same scene (with the furniture slightly rearranged) for comparison.
Figure 4.7: Instead of the two-dimensional colormapped images, the Kinect depth-mapping can be overlaid on the RGB image and exported to a point-cloud format. These point-cloud library (PCL) images contain real-world distances and allow for three-dimensional visualization on a computer screen. Examples are shown for an indoor scene (top) and an outdoor scene acquired in low-light conditions (bottom), viewed at an oblique angle to highlight the 3-D representation.
Figure 4.8: Images acquired outdoors using the Kinect IR sensor (left) and the corresponding depth mapped images (right) for a window (top) and tree (bottom) show the difficulties sunlight creates for the infrared sensor.
Figure 4.9: The infrared spectrum spans from just past the red end of the visible spectrum to wavelengths of 1 mm and is divided into five sub-divisions. The Kinect IR sensor is in the near-infrared, while the thermal imaging camera operates in the long-infrared regime.

Although the Kinect's depth mapping is of little use in outdoor environments during the day, it may still be useful outside at night. However, the point-cloud library representation will not work at night because it requires well-illuminated webcam image on which to overlay the depth information. An example of the usefulness of the Kinect depth mapping at night is shown in Figure 4.12, where the depth mapping highlights obstacles not visible in normal webcam images.

In summary, the Kinect's depth sensor will work outside under certain circumstances. Unfortunately, the Kinect's infrared sensor will not replace the more expensive thermal imaging camera to detect small signals from passive objects since it operates in the near-infrared regime instead of long-wave regime that is more sensitive to such signals.
Figure 4.10: The solar radiation spectrum has a distribution consistent with a 5525 K blackbody, but as light passes through the Earth’s atmosphere it is absorbed in specific bands by greenhouse gases and redistributed by Rayleigh scattering. The greater level of radiance in the near infrared regime as compared to the long-wave infrared explains why the Kinect’s sensor is saturated while used in direct sunlight while the thermal infrared camera is not. (Image from [87] and used under the terms of the GNU Free Documentation License)
Figure 4.11: As the sun emerges from behind a cloud and sunlight increases (top to bottom), the Kinect’s infrared sensor (left) becomes saturated and the Kinect is unable to construct a corresponding depth mapped image (right).
Figure 4.12: The Kinect depth mapping (A) works well in nighttime outdoor environments, detecting a light pole not visible in the illuminated RGB image (B). The image from the thermal camera (C) also shows the tree and buildings in the background, but has a smaller field of view and lower resolution than the raw image from the Kinect IR sensor (D) (images resized from original resolutions).
4.2.3 Audio

Our main interest in updating rMary is to see how acoustic sensors could be integrated into mobile robotics. Past work with rMary’s sibling rWilliam (Figure 4.13) investigated the use of air-coupled ultrasound in mobile robotics [88, 89, 90], as have others [91, 92, 93]. The high attenuation of ultrasound in air limits the use of ultrasound scanning for mobile robots.

Figure 4.13: A 50 kHz ultrasound scanner mounted on rWilliam in able to detect objects at close range.

Instead, we wish to study the use of low-frequency acoustic echolocation for mobile robots. This is similar in principle to how bats navigate, though at much lower frequencies and with much lower amplitude signals. A similar use of this technology is found in the Sonar ruler app for the iPhone that attempts to measure distances
using the speaker and microphone, with mixed results [94]. Using signals in the audible range reduces the attenuation, allowing for propagation over useful distances. However, there is more background noise in the audible frequency range, requiring the use of coded excitation signals and sophisticated signal processing techniques to find the reflected signal in inherently noisy data.

One way to ensure that the reflected signal is primarily backscattering from the target rather than clutter (unwanted reflections from the environment) is to create a tightly spatially-controlled beam of low-frequency sound using an acoustic parametric array. We can also use insights gleaned from simulations to improve the analysis methods. Chapter 7 discusses in detail a method of simulating the propagation of the nonlinear acoustic beam produced by the acoustic parametric array and its scattering from targets.

The properties of the acoustic parametric array have been studied in depth [95, 96, 97, 98, 99] and has been used for area denial, concealed weapons detection, and nondestructive evaluation [100, 101, 102]. In brief, the parametric array works by generating ultrasonic signals at frequencies \( f_1 \) and \( f_2 \), whose difference is in the audible range. As these signals propagate, the nonlinearity of air causes self-demodulation of the signal, creating signals at the sum \( (f_1 + f_2) \) and difference \( (f_2 - f_1) \) frequencies. Since absorption is proportional to the square of frequency, only the difference frequency remains as the signal propagates away from the array (Figure 4.14).

The acoustic parametric array allows for tighter spatial control of the low-frequency sound beam than a standard loudspeaker of the same size. Directivity of a speaker depends on the ratio of the size of the speaker to the wavelength of sound produced, with larger speakers able to create more directive low-frequency sound. Line arrays (speakers arranged in a row) are the best traditional way to create directional low-
Figure 4.14: The acoustic parametric array creates signals at two frequencies $f_1$ and $f_2$ in the ultrasonic range (pink shaded region). As the signals propagate away from the parametric array, the nonlinearity of air allows the signals to self-demodulate, creating signals at the sum and difference frequencies. Because attenuation is proportional to the square of frequency, the higher-frequency signals attenuate more quickly, and after several meters only the audible difference frequency remains.
frequency sound, but can take up a great deal of space [103]. Using nonlinear acoustics, the acoustic parametric array is able to create directional low-frequency sound in a normal-sized speaker, as shown in Figure 4.15.

For our tests, we have mounted the Sennheiser Audiobeam parametric array to rMary, with power supplied directly from rMary's battery. This commercially available parametric array uses a 40 kHz carrier signal to produce audible sound pressure levels of $75 \pm 5$ dB at a distance of 4 m from the face of the transducer. The echolocation signals we use are audible in front of the transducer at distances exceeding 50 m in a quiet environment, but would not necessarily be obtrusive to pedestrians passing through the area and are easily masked by low levels of external noise.

To record the backscattered echolocation signal, as well as ambient noise from our environment, we use the four channel microphone array included in the Kinect. This array is comprised of four spatially separated high-quality capsule microphones with a sampling rate of 16 kHz. The array separation is not large enough to allow implementation of beamforming methods at distances of interest here. The low sampling rate means that acoustic signals are limited to a maximum frequency content of 8 kHz.

Audio data recorded by the Kinect microphone array was compared to data recorded using a parabolic dish microphone (Dan Gibson EPM, 48 kHz sampling rate), whose reflector dish directs sound onto the microphone. Figure 4.16 shows that the microphones used in the Kinect actually perform better than the parabolic dish microphone [104, 105]. All data used in our subsequent analysis is recorded with the Kinect array.

This low-frequency acoustic echolocation sensor system is discussed in detail in Chapter 6, where vehicles are detected and classified into groups based on frontal profile at distances of 50 m using only the acoustic backscatter signals.
Figure 4.15: The acoustic beam created by the acoustic parametric array has a consistently tighter beam pattern than the much physically larger line array at low frequencies and fewer sidelobes at higher frequencies.
Figure 4.16: Even though the four-channel Kinect microphone array has tiny capsule microphones that only sample at 16 kHz, they provide a cleaner signal than the parabolic dish microphone with a 48 kHz sampling rate.

### 4.2.4 Radar

The final sensor currently on rMary is a coffee-can radar. A collaboration between MIT and Lincoln Labs in 2011 produced a design for a low-cost radar system that uses two metal coffee cans as antennas [106]. Simple amplifier circuits built on a breadboard power low-cost modular microwave (RF) components to send and acquire signals through the transmit (Tx) and receive (Rx) antennas. The entire system is powered by 8 AA batteries, which allows easy portability and the total cost of components is less than USD $350. Our constructed coffee-can radar is shown in Figure 4.17.

The signal processing requirements of the coffee can radar system are reduced by using a frequency modulated continuous wave (FMCW). In this setup, the radar
transmits an 80MHz chirped waveform centered at 2.4Ghz (in the ISM band). The same waveform is then used to downconvert, or 'de-chirp' the signal so that the residual bandwidth containing all the information is small enough to digitize with a sound card. This information is saved in .wav files and analyzed in Matlab.

The system as originally designed has 10 mW Tx power with an approximate maximum range of 1 km and can operate in one of three modes: Doppler, range, or Synthetic Aperture Radar (SAR). In Doppler mode the radar emits a continuous-wave signal at a given frequency. By measuring any frequency shifts in this signal, moving objects are differentiated from stationary ones. Images from this mode show an object's speed as a function of time. In ranging mode, the radar signal is frequency modulated, with the magnitude of this modulation specifying the transmit bandwidth. This allows the imaging of stationary or slowly-moving objects, and the resulting images show distance from the radar (range) as a function of time. SAR imaging is basically a set of ranging measurements acquired over a wide area to create a three-dimensional representation of the radar scattering from a target [107, 108, 109].

While SAR imaging has the greatest potential application in mobile robotics, since
the robotic platform is already moving over time, we will look at ranging measurements in our feasibility tests of the radar. Figure 4.18 shows a ranging measurement of three vehicles approaching rMary with initial detection of the vehicles at a distance of 70 m. Since the ranging image is a colormapped plot of time versus range, the speed of approaching vehicles can also be calculated directly from the image data.

These measurements demonstrate the feasibility of this low-cost radar as a long-range sensor for mobile robotics. Since the radar signal is de-chirped to facilitate processing with a computer sound card, these measurements may not contain information about scattering from the object, unlike the acoustic echolocation signal discussed in Chapter 5. However, radar ranging measurements could provide an early detection system for a mobile robot, detecting objects at long range before other sensors are used to classify the object. This detection distance is dependent upon a number of parameters, most important of which is the availability of line-of-sight to the target.

The rMary platform allows us to investigate the capabilities and limitations of a number of low-cost sensors in unstructured outdoor environments. A combination of short- and long-range sensors provides a mobile robot with the most useful information about its environment. Previous work focused on passive thermal infrared and air-coupled ultrasound as possible short-range sensor modalities. Our work looked at the suitability of the Microsoft Kinect as a short-range active infrared depth sensor, as well as the performance of a coffee can radar and acoustic echolocation via acoustic parametric array as long-range sensors for mobile robotics. While the low-cost depth sensor on the Microsoft Kinect is of limited use in outdoor environments, the coffee can radar has the potential to provide low-cost long-range detection capability. In addition, the Kinect microphone array can be paired with an acoustic parametric array to provide high-quality acoustic echolocation measurements.
Figure 4.18: The ranging image acquired using a coffee-can radar shows three vehicles approaching Mary. The vehicles’ speed can be calculated from the slope of the line.
Chapter 5

Acoustic echolocation from a mobile robot

In Chapter 4 we discussed a number of possible sensor modalities that could be combined to give a walking-speed autonomous robot situational awareness in unstructured environments. In particular, acoustic echolocation in the audible range using an acoustic parametric array and Kinect microphone array stood out as a possible new modality that could detect and classify objects at long distances. In this chapter we will investigate this acoustic echolocation sensor by attempting to detect and classify oncoming vehicles at distances of 50 m using only the backscattered acoustic data.

5.1 Vehicle classification

Attempts at remote vehicle classification began in the 1970s [110, 111] using radar [112], seismic activity [113], computer vision [114, 115], and the acoustic pass-by signature as detection modalities. Thomas and Wilkins attempted to classify vehicles by their acoustic spectrum using cepstrum analysis in 1972, but such features did not
provide enough separation between the classes [116]. Nooralahiyan et al. were more successful in 1997, achieving 84% accuracy for vehicles at an urban road site [117, 118]. More recently, Bao et al. use a vehicle detection algorithm based on bispectral entropy to detect running vehicles at distances of 1 km, but do not attempt classification of vehicles [119]. Guo et al. try to improve classification accuracy of acoustic pass-by measurements by using information fusion techniques to separate the internal sound production from sound produced by a vehicle's external parts [120].

These acoustic approaches have also been extended to other noise-generating vehicles. Averbuch et al. classified both land vehicles and boats based on the energy distribution of their acoustic signatures [121, 122]. Quaranta and Dimino extended this analysis to aircraft [123]. A number of other papers use alternative methods of analyzing the acoustic signal with varying levels of success [124, 125, 126].

Up to this point, acoustic detection and classification of vehicles has mostly been limited to analysis of the emitted acoustic signal, described in more detail in a review by Braun et al. as it relates to regulation in some European countries by international standard ISO 362 [127]. Changes in vehicle construction, including the recent trend toward quieter electric vehicles, as well as changes in road surfaces over time are among a number of complicating factors that make analysis of the vehicle pass-by signature difficult. Environmental noise and the difficulty in singling out pass-by signals on large, multi-lane roads further complicate the process.

Instead, we will use an active acoustic signal that can be targeted at a specific vehicle to classify oncoming vehicles. Similar to how bats navigate and track food, this acoustic echolocation can detect and classify vehicles at distances of 50 m or greater.
5.2 Acquiring acoustic echolocation data

We use the acoustic parametric array to propagate an acoustic signal from rMary to a passing vehicle and determine the type of vehicle from the reflected (backscatter) signal (Figure 5.1). Vehicles are divided into 7 classes based on size and frontal profile: car (c), sport utility vehicle (s), van (v), truck (t), bus (b), motorcycle (m), and other (o).

Figure 5.1: As a vehicle approaches rMary, an acoustic signal is transmitted from the forward-looking parametric array (not visible here), scatters from the oncoming target vehicle, and the reflected signal is recorded with the large parabolic dish microphone. This microphone has since been replaced by the Kinect microphone array.

The parametric array, previously described in detail in Chapter 4, allows us to create a tightly controlled beam of low-frequency sound that is able to propagate long distances. Previous work in robotic echolocation using 50 kHz sonar showed
how quickly ultrasonic signals attenuate in air due to absorption, a problem that is 
mitigated by using audible acoustic signals [88]. The trade-off is increased background 
oise levels in this range, further complicating detection. This is especially true for 
a roadside environment with pedestrian and vehicular noise sources in the audible 
range.

Initially the audio data was recorded with a parabolic dish microphone mounted 
atop rMary, providing single-channel audio data at 48 kHz. We soon discovered that 
the Kinect’s built-in four-channel microphone array allowed for high-quality recording 
with much less ambient noise, and discontinued use of the parabolic dish microphone. 
The Kinect microphone array is restricted to a 16 kHz sampling rate, allowing accurate 
reproduction of sounds with frequencies under 8 kHz. Intelligent restriction of the 
upper frequency limit of our transmitted signal to 4kHz allows downsampling of the 
recorded acoustic data by a factor of two without loss of important information.

A number of transmitted signals were tested and will be discussed in more detail 
later. In general, the signal was a series of 1-4 kHz linear chirps with defined pulse 
length and spacing between pulses. These are referred to in the format XXX-YYY, 
where XXX represents the pulse length in ms and YYY represents the spacing between 
pulses, also in ms. For example, a 5-750 signal contains a 5 ms long linear chirp, 
followed by 750 ms of silence. This sequence repeats continuously.

The first measurement method was to record in a continuous loop while the 
transmitted signal was broadcast. Spoken audio cues were used to indicate the 
vehicle’s class when it passed a specific distance marker. The recorded data was later 
manually marked in Audacity and cut into shorter 4-5 second bits using an automated 
script, so that each new file only contained data from a single vehicle. An example of 
this manual labeling is shown in Figure 5.2.

All of the 250-500, 250-1000, 10-750, and 5-750 data was recorded in this continuous
manner. The addition of a radar to rMary required switching to a measurement scheme where data from each vehicle is more clearly separated. In this scheme a single timestamped folder contains the 4-channel Kinect audio recording, radar data, and Kinect RGB snapshot of a single vehicle. A simple text file with these associated timestamps/vehicle classes allows for automated compilation of data. This single-measurement method was used for all 100-900 data. Current data compilation scripts allow use of all measured data; the two measurement schemes are indistinguishable from a data analysis perspective.

A small Linux netbook was used to acquire all data. The transmitted signal was played through the computer's on-board sound card using sox, and custom software using the open-source OpenKinect drivers (libfreenect) allowed the Kinect to simultaneously record audio data and RGB-D (depth) video [128]. The entire process was controlled using bash scripts listed in Appendix 8.2.

This measurement method doesn't ensure that a vehicle is always at a precise distance away from rMary for each measurement, since there is some human decision about when a vehicle passes the distance marker as well as physical variation of when the transmitted signal actually arrives at the vehicle. Determining the distance of an approaching vehicle is not our focus here but rather extracting information about what type (class) of vehicle is approaching.
We must make our analysis range-independent. The most important consideration is aligning the individual signals in time, which will be discussed in detail later. More careful alignment of signals in the measurement phase could be possible using precise distance-triggered measurement setups, but require consideration of other complicating environmental factors, such as the effect of temperature and humidity on the speed of sound in air. One possible solution is to use the coffee can radar described in Chapter 4 to measure the distance to an oncoming vehicle before using acoustic echolocation to classify the vehicle. Due to the precision required in our feature selection methods, further alignment of the signals would be necessary even if these methods were followed.

Data was collected from stationary vehicles in a parking lot and approaching vehicles on a two-lane non-divided street with a speed limit of 35 mph. Data from stationary vehicles allows us to optimize both the data collection and analysis routines at short distances in a much quieter environment, as well as to investigate the effect of reflections from vehicles at a range of oblique angles.

Data from approaching vehicles was acquired from more than 40 sorties over a year-long period. Data was collected over a range of environmental conditions, which were carefully recorded from a local weather station. Due to technical limitations of the hardware, no measurements were taken during active precipitation events (i.e., rain or snow) or when the air temperature fell below 20 ° F. To avoid introducing an site-specific bias in our dataset, data from oncoming vehicles was acquired at a number of locations shown in Figure 5.3. Table 5.1 shows the vehicle distance, transmitted signal, and dates data was acquired for each location.

Much of this data was obtained with rMary facing the vehicles at a very slight angle. This was necessary so that rMary could be parked on the sidewalk and not impede the normal flow of traffic. Locations 7 and 8 allow orthogonal alignment.
Table 5.1: The vehicle distance, transmitted signal, and dates which data was acquired is given below for each location shown in Figure 5.3.

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance (m)</th>
<th>Signal</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>250-500</td>
<td>24 Jan</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>250-500</td>
<td>25 Jan</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26 Jan</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>250-1000</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>250-1000</td>
<td>02 Feb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 Feb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13 Mar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 Mar</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>250-1000</td>
<td>15 Mar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 Mar</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>250-1000</td>
<td>22 Mar</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>250-1000</td>
<td>27 Mar</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>250-1000</td>
<td>10 Apr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>19 Apr</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>250-500</td>
<td>31 May</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>250-1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>06 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>08 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10-750</td>
<td>13 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11 Jul</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12 Jul</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5-750</td>
<td>15 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>18 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>19 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>27 Jun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>06 Jun</td>
</tr>
<tr>
<td></td>
<td>100-900</td>
<td></td>
<td>12 Feb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 Feb</td>
</tr>
<tr>
<td>108</td>
<td></td>
<td></td>
<td>27 Feb</td>
</tr>
</tbody>
</table>
Figure 5.3: Acoustic echolocation measurements from oncoming vehicles was acquired at a number of locations in Williamsburg, VA. Acquiring data from a number of different locations in different environmental conditions is critical to creating a diverse dataset from which robust classifiers can be trained. Head-on data is acquired from locations 7 & 8 and all radar data is acquired at location 8. Arrows represent the direction in which rMary was facing.

of the parametric array and the oncoming vehicle, and data from these locations is referred to as 'head-on' (HO). We will see later that there is no noticeable difference between results obtained in these two orientations. Table 5.2 shows the total amount of collected data for each class in both orientations.
Table 5.2: Number of measurements from individual oncoming vehicles for both a 'head-on' (HO) and oblique angle over 42 sorties. The seven vehicle classes are: car (c), sport utility vehicle (s), van (v), truck (t), bus (b), motorcycle (m), and other (o).

<table>
<thead>
<tr>
<th>Signal</th>
<th>HO</th>
<th>c</th>
<th>s</th>
<th>v</th>
<th>t</th>
<th>b</th>
<th>m</th>
<th>o</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>250-500</td>
<td>N</td>
<td>41</td>
<td>21</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>61</td>
<td>74</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>160</td>
</tr>
<tr>
<td>250-1000</td>
<td>N</td>
<td>589</td>
<td>410</td>
<td>52</td>
<td>28</td>
<td>20</td>
<td>2</td>
<td>6</td>
<td>1107</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>552</td>
<td>515</td>
<td>55</td>
<td>42</td>
<td>27</td>
<td>3</td>
<td>3</td>
<td>1197</td>
</tr>
<tr>
<td>100-900</td>
<td>Y</td>
<td>94</td>
<td>88</td>
<td>13</td>
<td>7</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>219</td>
</tr>
<tr>
<td>10-750</td>
<td>Y</td>
<td>191</td>
<td>190</td>
<td>22</td>
<td>18</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>431</td>
</tr>
<tr>
<td>5-750</td>
<td>Y</td>
<td>520</td>
<td>501</td>
<td>52</td>
<td>32</td>
<td>15</td>
<td>5</td>
<td>1</td>
<td>1126</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>2048</td>
<td>1799</td>
<td>211</td>
<td>133</td>
<td>89</td>
<td>14</td>
<td>16</td>
<td>4310</td>
</tr>
</tbody>
</table>

5.3 Initial data analysis

The first step in our analysis was to determine whether the reflected pulse is even present in the recorded signal. We also look at the effect of reflections from oblique angles, and whether this makes a difference in signal detectability. Our control of the transmitted signal is a vital part of our experimental procedure, allowing us to optimize the signal to improve detectability of the backscattered signal among the background noise.

5.3.1 Detecting a signal reflected from a wall

There are several ways to ensure that we are able to detect our signal among the background noise. The most obvious way is to simply increase the amplitude of the transmitted signal. This is not a good solution for our application, since autonomous robots should be as unobtrusive as possible. It also runs counter to our goal of minimizing the amount of noise added to the environment. The parametric array that we are using (Sennheiser Audiobeam) is rated for 75 ± 5 dB at a distance of 4 m.
This level is well within the background noise level of a typical passing vehicle. At 60 kph (approximately 35 mph), light cars have a mean emission level of 65dBA, while heavy trucks exceed 80 dBA [129].

Using an acoustic parametric array helps by concentrating the sound propagation to a small geographic area. This serves the dual purpose of allowing more of the sound energy to interrogate our target while reducing the amount of noise pollution added to the environment. However, generating a 75 dB audible signal from the parametric array requires a 140 dB signal at the face of the transducer. Even though the signal at the face of the transducer is entirely in the ultrasonic range and not audible to humans, such high levels have the potential to cause human hearing loss. The Sennheiser parametric array used here has a safety device that will silence the signal if objects are too close to the array.

A better way to improve signal detectability is to shape its frequency content, using coded signals. This will become especially useful when we try to detect the backscattered signal, allowing us to use cross-correlation methods. We have chosen to use variants of 1-4 kHz linear chirps as transmitted signals. The chirp length ranges from 5 to 250 ms, and spacing between chirps range from 500 to 900 ms. Longer pulse lengths have the advantage of delivering more energy to interrogate the target, but suffer in the ability to localize the reflected pulse in time. Extremely short chirps of 5 or 10 ms can more easily localize the reflected pulse in time, but deliver less energy to the target. A medium-length chirp (100ms) delivers more energy to the target than the short chirp signals while providing better localization in time than the long duration signals.

We can better visualize how the incident signal duration affects detectability of the backscattered signal by looking at the reflection of different signals from a flat brick wall. The wall is located near a busy road so the recorded signal contains background

111
noise comparable to that present in measurements from oncoming vehicles. The short (5 and 10 ms) linear chirps produce a sharp peak that is easy to localize in the time domain waveform, albeit only a short distances (i.e. 5 m). This is due to the lesser amount of energy contained in these signals. It is very important to point out that even though we can’t see a backscatter signal doesn’t mean that there is no useful information present. This point is further illustrated by looking at the time-frequency spectrogram representation of the same signal, where the reflected signal is obvious at greater distance (Figures 5.4 and 5.5).

By using a longer chirp signal (250 ms) with more energy content, we can see a clear signature in the time domain waveform at short distances and in the spectrogram at distances exceeding 50 m (Figure 5.4).

5.3.2 Data from oblique angles

Data was also collected from stationary vehicles at various angles. Though reflected sound is clearly audible at the complementary angle, enough is backscattered to the Kinect array to clearly see in the spectrogram even at an extreme angle of 60° (Figure 5.7).

These preliminary tests tell us several important things:

- Reflections at long distances are more easily detectable using longer-length chirp signals. The trade-off is localization in time, which is less of a worry since our application is classification rather than accurate distance tracking.

- Time-frequency representations of the data allow us to use information about the frequency content of the reflected signal to improve detectability. Other time-frequency-like representations may help even more, at the cost of abstraction.

- Reflections from vehicles seems to not be strongly influenced by a precise incident
Figure 5.4: The individual backscatter reflections of a 5-750 pulse from a flat wall is visible in both the dB waveform and the spectrogram at 5 m (top), but is not visible at all at 20 m (bottom).
Figure 5.5: The individual backscatter reflections of a 10-750 pulse from a flat wall is visible in both the dB waveform and the spectrogram at 5 m (top), but is not visible at all at 20 m (bottom).
Figure 5.6: The individual backscatter reflections of a 250-1000 pulse from a flat wall is clearly visible in both the dB waveform and the spectrogram at 5 m (top), but is only visible in the spectrogram at a distance of 50 m (bottom).
Figure 5.7: The individual backscatter reflections of a 10-750 pulse from a van at 10 m are clearly visible in the spectrogram at an orientation of 0° (top), 30° (middle), and 60° (bottom).
angle. That is, measurements taken at a slight angle will still contain useful information.

- Apparent undetectability of a reflected signal, even in the spectrogram, should not be taken to mean that no useful information exists. As an example, we will present data that shows useful classification results using the short chirp signals at distances of 50 m.

Chapter 6 will use the knowledge acquired from these measurements to extract information from the reflected signal that will allow us to classify vehicles based solely on their backscattered acoustic reflection.
Chapter 6

Classification of oncoming vehicles using acoustic echolocation

6.1 Pattern classification

So far we know that our transmitted signal is present in the backscattered reflection from a target at distances exceeding 50 m. The hope is that this reflected signal contains useful information that will allow us to determine the type (class) of vehicle. Since we are using a coded signal we also expect that a time-frequency representation of the data will prove useful in this classification process. The next step is to use statistical pattern classification techniques as discussed in Chapter 3 to find that useful information in these signals to differentiate between vehicles of different classes. These analyses are written in parallel to run in Matlab on a computing cluster to reduce computation time (Appendix 8.2).
6.1.1 Compiling data

To more easily compare the large number of measurements from different classes we organize the measured data into structures. The greater than 4000 individual measurements we have collected are spread over 5926 files, including audio, radar, and image data organized in timestamped directories. Separate plaintext files associate each timestamp with its corresponding vehicle class. If we are to run our analysis routines on computing clusters, providing access to this more than 3.6 GB of original data becomes problematic. Instead we create smaller data structures containing only the information we require. These datasets range in size from 11-135 MB for 108-750 measurements and can easily be uploaded to parallel computing resources.

Much of the reduction in size is due to the fact that we only require access to the audio data for these tests and can eliminate the large image files. One additional reduction is accomplished by choosing a single channel of the 4-channel Kinect audio data to include in the structure. The array has a small enough spacing that all useful information is present in every channel, as seen in Figure 6.1. Resampling all data to the acceptable minimum rate allowed by the Nyquist-Shannon sampling theorem further reduces the size of the data structure.

Our goal is to differentiate between vehicle classes, so it is natural to create data structures divided by class. Since it doesn't make sense to compare data from different incident signals, we create these structures for a number of data groups. We have also allowed the option to create combination classes, for example vans, trucks, and buses are combined into the 'vtb' class. This allows vehicles with similar frontal profiles to be grouped together to create a larger dataset to train our classifier. When creating these structures, data is pulled at random from the entire set of possibilities. The data structure can contain either all possible data or equal amounts of data from each class (or combined class), which can help reduce classification errors due to unequal
Figure 6.1: Due to the close spacing of the microphone array on the Kinect, all four channels contain the same information. The parabolic microphone is mounted aft of the Kinect array, causing the slight delay visible here, and is much more sensitive to noise.

data distribution. It is also important to note that due to the difficulty of detecting individual reflections inside a signal, not every measurement inside the data structure is guaranteed to be usable. Tables 6.1 and 6.2 of the amount of data in each group only provide an upper limit on the number of usable measurements.

6.1.2 Aligning reflected signals

The first step in our pattern classification process is to align the signals in time. This is crucial to ensure that we are comparing the signals reflected from vehicles to each other, rather than comparing a vehicle reflection to a background measurement that contains no useful information.

Our control of the transmitted signal gives us several advantages that we can
Table 6.1: Maximum amount of data in each classification group (overlaps possible between groups).

<table>
<thead>
<tr>
<th>Group</th>
<th>c</th>
<th>s</th>
<th>v</th>
<th>t</th>
<th>b</th>
<th>m</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-750 HO</td>
<td>520</td>
<td>501</td>
<td>52</td>
<td>32</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10-750 HO</td>
<td>191</td>
<td>190</td>
<td>22</td>
<td>18</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>100-900 HO</td>
<td>94</td>
<td>88</td>
<td>13</td>
<td>7</td>
<td>16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>250-500 HO</td>
<td>61</td>
<td>74</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>250-500 NHO</td>
<td>41</td>
<td>21</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250-500 all</td>
<td>102</td>
<td>95</td>
<td>17</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>250-1000 HO</td>
<td>552</td>
<td>515</td>
<td>55</td>
<td>42</td>
<td>27</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>250-1000 NHO</td>
<td>589</td>
<td>410</td>
<td>52</td>
<td>28</td>
<td>20</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>250-1000 all</td>
<td>1141</td>
<td>925</td>
<td>107</td>
<td>70</td>
<td>47</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>250-comb HO</td>
<td>613</td>
<td>589</td>
<td>71</td>
<td>45</td>
<td>30</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>250-comb NHO</td>
<td>630</td>
<td>431</td>
<td>53</td>
<td>31</td>
<td>21</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>250-comb all</td>
<td>1243</td>
<td>1020</td>
<td>124</td>
<td>76</td>
<td>51</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.2: Maximum amount of data in each classification group when binned (overlaps possible).

<table>
<thead>
<tr>
<th>Group</th>
<th>c</th>
<th>s</th>
<th>vtbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-750 HO</td>
<td>520</td>
<td>501</td>
<td>99</td>
</tr>
<tr>
<td>10-750 HO</td>
<td>191</td>
<td>190</td>
<td>47</td>
</tr>
<tr>
<td>100-900 HO</td>
<td>94</td>
<td>88</td>
<td>36</td>
</tr>
<tr>
<td>250-500 HO</td>
<td>61</td>
<td>74</td>
<td>22</td>
</tr>
<tr>
<td>250-500 NHO</td>
<td>41</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>250-500 all</td>
<td>102</td>
<td>95</td>
<td>27</td>
</tr>
<tr>
<td>250-1000 HO</td>
<td>552</td>
<td>515</td>
<td>124</td>
</tr>
<tr>
<td>250-1000 NHO</td>
<td>589</td>
<td>410</td>
<td>100</td>
</tr>
<tr>
<td>250-1000 all</td>
<td>1141</td>
<td>925</td>
<td>224</td>
</tr>
<tr>
<td>250-comb HO</td>
<td>613</td>
<td>589</td>
<td>146</td>
</tr>
<tr>
<td>250-comb NHO</td>
<td>630</td>
<td>431</td>
<td>105</td>
</tr>
<tr>
<td>250-comb all</td>
<td>1243</td>
<td>1020</td>
<td>251</td>
</tr>
</tbody>
</table>
exploit in this analysis. First, since the frequency content of the transmitted signal is
known, we can apply a bandpass filter to the reflected signal to reduce noise at other
frequencies. In some cases this will highlight reflections that were previously hidden
in the noise floor, allowing for automated peak detection.

More often, however, the backscattered reflection remains hidden among the noise
even after a bandpass filter is applied. In this case we obtain better results using peak
detection on the envelope signal.

To create this signal, we take our original signal $f(x)$ (which has already been
bandpass-filtered) and take the absolute value of its Hilbert transform $|\hat{f}(x)|$. This
is the analytic signal, which discards the negative frequency components of a signal
created by the Fourier transform in exchange for dealing with a complex-valued
function. The envelope signal is then constructed by applying a very lowpass filter to
the analytic signal. This process is shown in Figure 6.2.

In some cases, even peak detection on the envelope signal will not give optimal
results. Occasionally, signals will have a non-constant DC offset that complicates the
envelope signal. This can often be corrected by detrending (removing the mean) the
signal. A more pressing issue is that the envelope signal is not a reliable detection
method if reflections aren’t visible in the filtered signal. Even when peaks can be
detected in the envelope signal, they tend to be very broad. As a general rule, peak
detection is less sensitive to variations in threshold as the peak grows sharper. Adding
a step to the peak detection that finds the mean value of connected points above a
certain threshold ameliorates this problem, but since the peak widths of the envelope
signal are not uniform, finding the same point on each reflected signal becomes an
issue. Some of these issues are highlighted in Figure 6.3.

Instead, we can exploit another feature of our transmitted signal – its shape. All
of our pulses are linear frequency chirps which have well-defined characteristics and,
Figure 6.2: The envelope signal is created by taking the original bandpass-filtered data (top), creating the analytic signal (middle), and applying a very lowpass filter ($5^{th}$ order Butterworth, $f_c = 20$ Hz) (bottom).

more importantly, maintain a similar shape even after they reflect from a target (demonstrated in Chapter 7). By taking the cross-correlation of our particular transmitted signal and the reflected signal and accounting for the time shift inherent to the process, a sharp peak that can easily found by an automated peak detection algorithm is created at time point where the reflected signal begins.

Peak detection in any form requires setting a threshold at a level which reduces the number of false peaks detected without disqualifying actual peaks. This is a largely trial-and-error process and can easily introduce a human bias into the results. Setting
Figure 6.3: Even for multiple measurements from the same stationary vehicle, the envelope (red) has difficulty consistently finding peaks unless they are obviously visible in the filtered signal (blue). The shifted cross-correlation (green) doesn't have this limitation.
the threshold as a percentage of the signal's maximum value will also improve the performance of the peak detection.

Several problems with the automated peak detection are clearly visible in Figure 6.4, where a detection level set to 70% of the maximum value will only detect one of the three separate reflections that a human would identify. Although we could adjust the threshold level to be more inclusive, it would also increase the rate of false detection and add more computational load to filter these out. Adjustment of the threshold is also not ideal as it can add a human bias to the procedure.

Another issue is due to the shape of the correlated waveform, caused by a vehicle noise increasing as it nears the microphone. The extra noise in the first part of the signal is above the detection threshold and will lead to false detection. This is an easier problem to solve – our algorithm will reject any peaks that are not separated by large enough distance. A separation distance of half the length of the cut signals reduces the rate of false detection.

It is also important to note the position of the sensors on the robotic platform at this point. If we were using a regular loudspeaker, the transmitted pulse would be recorded along with the reflected signal and the cross-correlation would detect both signals, complicating the detection process. Mounting the microphone array behind the speaker could help, but care would have to be taken with speaker selection. Since the acoustic parametric array transmits an ultrasonic signal, the audible signal is only audible at greater distances than the position of the microphone array.

The three detection methods (filtered signal, envelope signal, and shifted cross-correlation) are summarized in Figure 6.5, which uses the simplified situation of data from a stationary vehicle at 25 m to illustrate all three methods. For our analysis we will use the shifted cross-correlation to align the signals in time.
Figure 6.4: Individual reflections aren't visible in the bandpass filtered signal from an oncoming vehicle at 50m (top) or in its detrended envelope signal (middle). Cross-correlation of the filtered signal and the 100-900 transmitted pulse (bottom) show clear peaks at the beginning of each reflected pulse, which can be used in an automated detection algorithm.
Figure 6.5: For the simpler case of a stationary van at 25 m, the backscatter reflection is clearly visible once it has been bandpass filtered (blue). Automated peak detection may correctly find the peaks of the envelope signal (red), but is much more sensitive to the threshold level than the shifted cross-correlation signal (green) due to the sharpness of its peaks.

6.1.3 Feature creation with DWFP

Preliminary tests in Chapter 5 suggest that a time-frequency representation of the backscatter signal will offer the most useful analysis. A number of methods of joint time-frequency analysis are presented in Chapter 2 and will not be discussed in depth here.

We will use the Dynamic Wavelet Fingerprint (DWFP) to represent our time domain waveforms in a time-frequency domain. This analysis has proven useful in past work to reveal subtle features in noisy signals [13, 14, 15, 16] by transforming a one-dimensional, time domain waveform to a two-dimensional time-scale image. An
example of the DWFP process is shown in Figure 6.6 for real-world data.

Figure 6.6: A one-second long acoustic signal reflected from a bus (top) is filtered (middle) and transformed into a time-scale image that resembles a set of individual fingerprints (bottom). This image is a pre-segmented ternary image that can easily be analyzed using existing image processing algorithms.

The main advantage of the DWFP process is that the output is a pre-segmented image that can be analyzed using existing image processing techniques. We implement these libraries to create a number of one-dimensional parameter waveforms that describe the image, and by extension our original signal. This analysis, which is described in more detail in Chapter 2 yields approximately 25 parameter waveforms.

As an overview, our feature extraction process takes a time domain signal and applies a bandpass-filter. A pre-segmented fingerprint image is created using the DWFP process, from which a number of one-dimensional parameter waveforms are extracted. In effect, our original one-dimensional time domain signal is now represented by multiple parameter waveforms. Most importantly, the time axis is maintained throughout this process so that features of the parameter waveform are directly correlated to events in the original time domain signal. A visual representation of the process is shown in Figure 6.7.

The user has control of a large number of parameters in the DWFP creation and
Figure 6.7: A one-second long 100-900 backscatter signal is bandpass filtered and converted to an ternary image using the DWFP process. Since the image is pre-segmented it is easy to apply existing image analysis techniques and create approximately 25 one-dimensional parameter waveforms that describe the image. Our original signal is now represented by these parameter waveforms, three examples of which are shown here (ridge count, filled area, and orientation). Since the time axis is maintained throughout the entire process, features in the parameter waveforms are directly correlated to events in the original time domain image.
feature extraction process, which greatly affect the appearance of the fingerprint images, and thus the extracted features. The parameters that most affect the fingerprint image are the wavelets used for pre-filtering and performing the continuous wavelet transform to create the DWFP image. A list of candidate wavelets is shown in Table 6.3. However, there is no way to tell \textit{a priori} which combination of parameters will create the ideal representation for a particular application. We use a computing cluster to run this process in parallel for a large number of parameter combinations, combined with past experience with analysis of DWFP images to avoid an entirely brute force implementation.

Table 6.3: List of usable wavelets. For those wavelet families with multiple representations (db, sym, and coif), the default value used is shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Matlab name</th>
<th>Prefiltering</th>
<th>Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haar</td>
<td>haar</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Daubechies</td>
<td>db3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Symlets</td>
<td>sym5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Coiflets</td>
<td>coif3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Meyer</td>
<td>meyr</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Discrete Meyer</td>
<td>dme</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Mexican hat</td>
<td>mexh</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Morlet</td>
<td>morl</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

6.1.4 Intelligent feature selection

Now that each original one-dimensional backscatter signal is represented by a set of continuous one-dimensional parameter waveforms, we need to determine which will best differentiate between different vehicles. The end goal is to create a small dimensional feature vector for each original backscatter signal which contains the value of a parameter waveform at a particular point in time. By choosing these time
points correctly, we have created a new representation of the signal that is much more information-dense than the original signal. This feature vector completely describes the original signal and can be used in statistical pattern classification algorithms to classify the data in seconds. More background on the feature selection process is discussed in Chapter 3.

For this analysis we are using a variant of linear discriminant analysis to find the points in time where the parameter waveform has the greatest separation between different classes, but also where signals of the same class have a small variance. For each parameter waveform, all of those signals from a single class are averaged to create a mean and corresponding standard deviation signal. Comparing the mean signals to each other and keeping a running average of the difference allows us to create an overall separation distance signal \( \delta \), while a measure of the variance between signals of the same class comes from the maximum standard deviation of all signals \( \sigma \). Instead of using iterative methods to simultaneously maximize \( \delta \) and minimize \( \sigma \), we create a ratio signal \( \rho = \frac{\delta}{\sigma} \) and find its maxima (Figure 6.8).

We save the time point and value of \( \rho \) of the top 5-10 points for each extracted feature. When this process has been completed for all parameter waveforms, this list is sorted based on decreasing \( \rho \) and reduced to the top 25-50 points, keeping track of both points and feature name. Restricting the process in this manner tends to create a feature vector with components from many of the features, as shown in Figure 6.9. The number of top points saved for both steps is a user parameter, shown in Table 6.4 and restricted to mitigate the curse of dimensionality.

Feature vectors can then be created for each original backscatter signal by taking the value of the selected features at the given points. This results in a final, dense feature vector representation for each original signal.

The entire pattern classification process for data from three classes is summarized
Figure 6.8: For each of the parameter waveforms (FilledArea shown here), a mean value is created by averaging all the measurements of that class (top). The distance between classes is quantified by the separation distance (middle left) and tempered by the intraclass variance, represented by the maximum standard deviation (middle right). The points that are most likely to separate the classes are shown as the peaks of the joint separation curve (bottom).
Figure 6.9: The list of top features selected for 100-900 (left) and 250-500 (right) datasets illustrate how features are chosen from a variety of different parameter waveforms.

Table 6.4: List of user parameters in feature selection.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peakdetect</td>
<td>joint, separate</td>
<td>Method to choose top points</td>
</tr>
<tr>
<td>viewselected</td>
<td>binary switch</td>
<td>View selected points</td>
</tr>
<tr>
<td>selectnfeats</td>
<td>$Z^+$</td>
<td>Keep this many top points for each feature</td>
</tr>
<tr>
<td>topnfeats</td>
<td>$Z^+$</td>
<td>Keep this many top points overall</td>
</tr>
</tbody>
</table>

in Figure 6.10.
Figure 6.10: For each class, every individual measurement is filtered and transformed to a fingerprint image, from which a number of parameter waveforms are extracted. For each of these parameter waveforms, an average is created for each class. A comparison of these average waveforms finds the points that are best separated between the classes, and the feature vector is compiled using the values of the parameter waveform at these points. This image diagrams the process for sample data from three classes (blue, green, and red).
6.1.5 Statistical pattern classification

The final step in our analysis is to test the ability of the feature vector to differentiate between vehicle classes using various pattern classification algorithms. This is often the most time consuming step in the process, but since we have used intelligent feature selection to create an optimized and small feature vector, this is the fastest step of the entire process here, and can be completed in seconds on a desktop machine. Of course, we have simply shifted the hard computational work that requires a computing cluster to the feature selection step. That is not to say that there are no advantages to doing the analysis this way – having such small feature vectors allows us to easily test a number of parameters of the classification.

Before we can run pattern classification routines we must separate our data into training and testing (or validation) data sets. By withholding a subset of the data for testing the classifier's performance, we can eliminate any 'cheating' that comes from using training data for testing. We also use equal amounts of data from each class for both testing and training to eliminate bias from unequal-sized data sets, as discussed in Chapter 3.

Our classification routines are run in Matlab, using a number of standard classifiers included in the PRTools toolbox [62]. Because of our small feature vectors and short classification run time, we run the pattern classification many times, randomizing the data used for the testing and training for each run. This gives us an average classification performance and allows us to use standard deviation of correct classifications as a measure of classification repeatability. While this single-valued metric is useful in comparing classifiers, more detailed information about classifier performance will come from the average confusion matrix. For \( n \) classes, this is an \( n \times n \) matrix that plots the estimated class against the known class. The confusion matrix for a perfect classification would resemble the identity matrix, with values of 1 on the diagonal and
0 elsewhere. An example of a confusion matrix is shown in Figure 6.11.

![Confusion Matrix Example](image)

**Figure 6.11:** This example of a real confusion matrix shows good classification performance, with high values on the diagonal and low values elsewhere. The confusion matrix allows more detailed visualization of the classification performance for specific classes than the single-value metric of overall percent correct. For example, although this classification has a fairly high accuracy of 86% correct, the confusion matrix shows that most of the error comes from misclassifying SUVs into the van/truck/bus class [udc classifier, 20 runs].

### 6.2 Results

We illustrate the use of the pattern classification analyses on data collected from both stationary and oncoming vehicles. Due to the similar frontal profiles of vans, trucks, and buses, and to mitigate the small number of observations recorded from
these vehicles, we will create a combined class of these measurements. The classes for classification purposes are then ‘car’, ‘SUV’, and ‘van/truck/bus’. For this three-class problem, a classification accuracy of greater than 33% means the classifier is performing better than random guessing.

6.2.1 Proof-of-concept: Acoustic classification of stationary vehicles

We begin our analysis with data collected from stationary vehicles. The first test is a comparison of classification performance when observations come from a single vehicle as compared to multiple vehicles. Multiple observations were made from vehicles in a parking lot at distances between 5 and 20 m. The orientation is approximately head-on (orthogonal) but with slight repositioning after every measurement to construct a more realistic dataset. The classification accuracy shown in Figure 6.12 validates our expectation that classification performs better when observations are exclusively from a single vehicle rather than from a number of different vehicles.

We see that reasonable classification accuracies can be achieved even when the observations come from multiple vehicles. Optimizing the transmitted signal and recognizing the importance of proper alignment of the reflected signals will help improve classification performance, as seen in Figure 6.13. Here, data is collected from multiple stationary vehicles at a range of short distances between 10 and 25 m using both a 10-750 and 250-1000 transmitted signal. Classification performance seems slightly better for the shorter chirp-length signal, but real improvement comes from ensuring the signals are aligned in time. For this dataset, alignment was ensured by visual inspection of all observations in the 250-1000 dataset. This labor-intensive manual inspection has been replaced by cross-correlation methods described earlier.
Figure 6.12: The first attempt to classify vehicles based on their backscattered acoustic reflection from a 250-1000 transmitted signal shows very good classification accuracy when all observations come from a single vehicle at 20 m (top). When observations come from multiple vehicles at 5 m, the classification accuracy is much lower but still better than random guessing (bottom) [knn classifier, 10 runs].
in the analysis of data from oncoming vehicle.

While these initial tests exhibit poorer classification performance than the data from oncoming vehicles, it is worth noting that these datasets consist of relatively few observations and are intended as a proof-of-concept. These stationary datasets were used to optimize the analysis procedure for the more interesting data from oncoming vehicles. For example, alignment algorithms weren't yet completely developed, and the k-nearest neighbor classifier used to generate the above confusion matrices has proven to have consistently worse performance than the classifiers used for the results shown from oncoming vehicles. Nevertheless, we see that better-than-random-guessing classification accuracy is possible using only the acoustic echolocation signal.

6.2.2 Acoustic classification of oncoming vehicles

Now that we have seen that it is possible to classify stationary vehicles using only the reflected acoustic echolocation signal, the more interesting problem is trying to classify oncoming vehicles at greater distances.

Since the DWFP feature creation process has a large number of user parameters, the first step was to find which few parameters will give us the best classification performance. This reduces the parameter space in our analysis and allows us to focus on more interesting details of the classification, such as the effect of the transmitted signal on the classification accuracy. Through previous work we have seen that the choice of wavelet in both the prefiltering and transform stage in the DWFP process cause the greatest change in the fingerprint appearance, and thus the features extracted from the fingerprint.

Table 6.5 shows the classification accuracy for different prefiltering wavelets, while Table 6.6 shows the classification accuracy for different transform wavelets. In both cases, the dataset being classified is from vehicles at 50m approaching head-on using
Figure 6.13: Observations from multiple stationary vehicles at distances of 10-25m shows approximately equal classification performance for both the 10-750 (left) and 250-1000 (right) transmitted signals. Manual inspection of the observations in the 250-1000 dataset to ensure clear visual alignment leads to markedly improved classification performance (bottom) [knn, 10 runs].
the 100-900 transmitted signal. The settings for other user parameters are: filtering at 5 levels, removing the first 5 details, 15 slices of thickness 0.03, and removing fingerprints that do not have a solidity in the range 0.3-0.6. The mean correct classification rate is created from 20 classification runs for each classifier. These are the parameters for the rest of our analysis unless noted otherwise.

The choice of prefiltering wavelet does not affect the classification accuracy much. The variance measure (given by the standard deviation of repeated classifications) is not shown, but is consistently around 0.07 for all classifiers. With this knowledge, there is no obvious preference of prefiltering wavelet and we chose coif3 for further analysis.

The choice of transform wavelet does seem to affect the classification accuracy somewhat more than the choice of the prefiltering wavelet. Still, the choice of classifier is by far the most important factor in classification accuracy. We select db3 as the default transform wavelet, with dmey and sym5 as alternatives.

From this analysis we can also select a few good classifiers for our problem. Pre-selecting classifiers violates the Ugly Duckling theorem, which states that we should not prefer one classifier over another, but because the underlying physical situation is similar between all of our datasets we are justified in selecting a small number of well-performing classifiers. We will use the top 5 classifiers from our initial analysis: nmsc, perlc, ldc, fisherc, and udc. The klldc and pcldc classifiers also performed well, but since they are closely related to ldc, we choose other classifiers for diversity and to provide a good mix of parametric and non-parametric classifiers.

Now more in-depth analysis of the effect that physical differences have on classification accuracy can be explored, using coif3 as a prefiltering wavelet and db3 as the transform wavelet, with the nmsc, perlc, ldc, fisherc, and udc classifiers.

Table 6.7 shows the datasets constructed for the following analyses. All datasets are
Table 6.5: A comparison of prefiltering wavelet (PW) choice on classification accuracy. The transform wavelet is db3. Data is from the 100-900 approaching vehicle dataset with a train/test ratio of 0.7 and classification into three classes (c, s, vtb). The differences in performance between classifiers falls within the measure of variance for a single classifier (not shown here for reasons of space). Since there seems to be no preferred prefiltering wavelet, future analysis will use coif3.

<table>
<thead>
<tr>
<th>PW</th>
<th>qdc</th>
<th>udc</th>
<th>ldc</th>
<th>kldc</th>
<th>pcldc</th>
<th>nmc</th>
<th>nmsc</th>
<th>logic</th>
<th>fisher</th>
<th>knnc</th>
<th>parzenc</th>
<th>parzencd</th>
<th>kernelc</th>
<th>perlc</th>
<th>svc</th>
<th>nusvc</th>
<th>treec</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>haar</td>
<td>0.40</td>
<td>0.81</td>
<td>0.83</td>
<td>0.78</td>
<td>0.79</td>
<td>0.66</td>
<td>0.87</td>
<td>0.75</td>
<td>0.82</td>
<td>0.56</td>
<td>0.63</td>
<td>0.73</td>
<td>0.59</td>
<td>0.86</td>
<td>0.67</td>
<td>0.70</td>
<td>0.51</td>
<td>0.70</td>
</tr>
<tr>
<td>db3</td>
<td>0.49</td>
<td>0.78</td>
<td>0.80</td>
<td>0.76</td>
<td>0.78</td>
<td>0.57</td>
<td>0.86</td>
<td>0.69</td>
<td>0.79</td>
<td>0.59</td>
<td>0.58</td>
<td>0.78</td>
<td>0.57</td>
<td>0.80</td>
<td>0.72</td>
<td>0.68</td>
<td>0.49</td>
<td>0.69</td>
</tr>
<tr>
<td>sym5</td>
<td>0.41</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.54</td>
<td>0.85</td>
<td>0.73</td>
<td>0.79</td>
<td>0.54</td>
<td>0.52</td>
<td>0.72</td>
<td>0.58</td>
<td>0.84</td>
<td>0.73</td>
<td>0.69</td>
<td>0.54</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>coif3</td>
<td>0.55</td>
<td>0.85</td>
<td>0.77</td>
<td>0.80</td>
<td>0.79</td>
<td>0.60</td>
<td>0.89</td>
<td>0.77</td>
<td>0.80</td>
<td>0.57</td>
<td>0.59</td>
<td>0.69</td>
<td>0.66</td>
<td>0.86</td>
<td>0.72</td>
<td>0.47</td>
<td>0.71</td>
<td>0.71</td>
</tr>
<tr>
<td>Average</td>
<td>0.46</td>
<td>0.81</td>
<td>0.80</td>
<td>0.79</td>
<td>0.59</td>
<td>0.87</td>
<td>0.73</td>
<td>0.80</td>
<td>0.56</td>
<td>0.58</td>
<td>0.73</td>
<td>0.60</td>
<td>0.84</td>
<td>0.71</td>
<td>0.69</td>
<td>0.50</td>
<td>0.70</td>
<td></td>
</tr>
</tbody>
</table>
Table 6.6: A comparison of transform wavelet (TW) choice on classification accuracy shows very similar classification performance for many wavelet choices. The prefiltering wavelet is coif3. Data is from the 100-900 approaching vehicle dataset with train/test ratio of 0.7 and classification into three classes (c, s, vtb). Due to space constraints, the variance is not shown.

<table>
<thead>
<tr>
<th>TW</th>
<th>qdc</th>
<th>udc</th>
<th>ldc</th>
<th>kldc</th>
<th>pldc</th>
<th>nmc</th>
<th>nmsc</th>
<th>logic</th>
<th>fisher</th>
<th>knnc</th>
<th>parzen</th>
<th>parzena</th>
<th>keernel</th>
<th>perl</th>
<th>svc</th>
<th>nusvc</th>
<th>treec</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>haard</td>
<td>0.45</td>
<td>0.68</td>
<td>0.73</td>
<td>0.67</td>
<td>0.68</td>
<td>0.42</td>
<td>0.81</td>
<td>0.59</td>
<td>0.66</td>
<td>0.47</td>
<td>0.47</td>
<td>0.67</td>
<td>0.54</td>
<td>0.77</td>
<td>0.61</td>
<td>0.63</td>
<td>0.48</td>
<td>0.61</td>
</tr>
<tr>
<td>db3</td>
<td>0.55</td>
<td>0.85</td>
<td>0.77</td>
<td>0.80</td>
<td>0.79</td>
<td>0.60</td>
<td>0.89</td>
<td>0.77</td>
<td>0.80</td>
<td>0.57</td>
<td>0.59</td>
<td>0.69</td>
<td>0.66</td>
<td>0.86</td>
<td>0.72</td>
<td>0.71</td>
<td>0.47</td>
<td>0.71</td>
</tr>
<tr>
<td>sym5</td>
<td>0.55</td>
<td>0.77</td>
<td>0.78</td>
<td>0.79</td>
<td>0.81</td>
<td>0.53</td>
<td>0.86</td>
<td>0.72</td>
<td>0.82</td>
<td>0.53</td>
<td>0.54</td>
<td>0.73</td>
<td>0.61</td>
<td>0.80</td>
<td>0.71</td>
<td>0.66</td>
<td>0.49</td>
<td>0.69</td>
</tr>
<tr>
<td>coif3</td>
<td>0.45</td>
<td>0.77</td>
<td>0.68</td>
<td>0.67</td>
<td>0.67</td>
<td>0.49</td>
<td>0.76</td>
<td>0.64</td>
<td>0.64</td>
<td>0.49</td>
<td>0.51</td>
<td>0.68</td>
<td>0.55</td>
<td>0.75</td>
<td>0.65</td>
<td>0.63</td>
<td>0.56</td>
<td>0.62</td>
</tr>
<tr>
<td>meyr</td>
<td>0.43</td>
<td>0.65</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.51</td>
<td>0.86</td>
<td>0.66</td>
<td>0.80</td>
<td>0.53</td>
<td>0.61</td>
<td>0.68</td>
<td>0.62</td>
<td>0.77</td>
<td>0.69</td>
<td>0.65</td>
<td>0.47</td>
<td>0.66</td>
</tr>
<tr>
<td>dmeys</td>
<td>0.43</td>
<td>0.84</td>
<td>0.82</td>
<td>0.80</td>
<td>0.82</td>
<td>0.50</td>
<td>0.88</td>
<td>0.81</td>
<td>0.80</td>
<td>0.47</td>
<td>0.50</td>
<td>0.71</td>
<td>0.54</td>
<td>0.88</td>
<td>0.71</td>
<td>0.62</td>
<td>0.52</td>
<td>0.69</td>
</tr>
<tr>
<td>mexh</td>
<td>0.47</td>
<td>0.64</td>
<td>0.76</td>
<td>0.77</td>
<td>0.73</td>
<td>0.55</td>
<td>0.78</td>
<td>0.63</td>
<td>0.76</td>
<td>0.57</td>
<td>0.51</td>
<td>0.63</td>
<td>0.61</td>
<td>0.73</td>
<td>0.64</td>
<td>0.63</td>
<td>0.44</td>
<td>0.64</td>
</tr>
<tr>
<td>morl</td>
<td>0.47</td>
<td>0.68</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.52</td>
<td>0.84</td>
<td>0.72</td>
<td>0.83</td>
<td>0.51</td>
<td>0.51</td>
<td>0.76</td>
<td>0.51</td>
<td>0.85</td>
<td>0.70</td>
<td>0.65</td>
<td>0.51</td>
<td>0.68</td>
</tr>
<tr>
<td>Average</td>
<td>0.47</td>
<td>0.73</td>
<td>0.77</td>
<td>0.76</td>
<td>0.76</td>
<td>0.51</td>
<td>0.83</td>
<td>0.69</td>
<td>0.76</td>
<td>0.52</td>
<td>0.53</td>
<td>0.69</td>
<td>0.58</td>
<td>0.80</td>
<td>0.68</td>
<td>0.65</td>
<td>0.49</td>
<td>0.70</td>
</tr>
</tbody>
</table>
constructed from data pulled at random from the overall datasets from that particular
signal type and contain an equal number of instances for each class. Most of the
datasets consist of three classes: car (c), SUV (s), and a combined van/truck/bus
(vtb), though a few datasets with data from all five classes: car (c), SUV (s), van (v),
truck (t), bus (b) were created to attempt this individual classification. Requiring an
equal number of instances from each class leads to small datasets, even after creating
the combined van/truck/bus class to mitigate this effect. In addition, not all of the
instances are usable due to the difficulty of detecting and aligning the signals. This
is especially true for the 250 ms signals.

Table 6.7: A survey of the datasets used in this analysis (in order of appearance)
shows the number of classes, total number of instances, and distance at which data
was acquired. The small size of the datasets is a direct result of requiring the datasets
to have an equal number of instances per class and the relatively few observations
from vans, trucks, and buses.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Classes</th>
<th>Instances</th>
<th>Distance (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-900 a</td>
<td>3</td>
<td>108</td>
<td>50</td>
</tr>
<tr>
<td>100-900 b</td>
<td>3</td>
<td>108</td>
<td>50</td>
</tr>
<tr>
<td>100-900 c</td>
<td>3</td>
<td>108</td>
<td>50</td>
</tr>
<tr>
<td>250-comb a</td>
<td>3</td>
<td>251</td>
<td>25, 30, 50</td>
</tr>
<tr>
<td>250-comb b</td>
<td>3</td>
<td>251</td>
<td>25, 30, 50</td>
</tr>
<tr>
<td>250-comb c</td>
<td>3</td>
<td>251</td>
<td>25, 30, 50</td>
</tr>
<tr>
<td>250-500</td>
<td>3</td>
<td>66</td>
<td>30, 50</td>
</tr>
<tr>
<td>250-1000</td>
<td>3</td>
<td>66</td>
<td>25, 25, 30, 60</td>
</tr>
<tr>
<td>250-comb HO</td>
<td>3</td>
<td>98</td>
<td>50</td>
</tr>
<tr>
<td>250-comb NHO</td>
<td>3</td>
<td>98</td>
<td>25, 30</td>
</tr>
<tr>
<td>5-750</td>
<td>3</td>
<td>108</td>
<td>50</td>
</tr>
<tr>
<td>10-750</td>
<td>3</td>
<td>108</td>
<td>50</td>
</tr>
<tr>
<td>100-900</td>
<td>5</td>
<td>35</td>
<td>50</td>
</tr>
</tbody>
</table>

We will first look at the influence of the train/test ratio on the classification
performance. Results for a sample glass dataset in Chapter 3 showed an increase

144
in classification accuracy as more data was used for training. Table 6.8 shows the classification accuracy as a function of train/test ratio for the same 100-900 dataset used in our earlier analysis of wavelets and classifiers shown in Tables 6.5 and 6.6. In general, the classifiers are able to perform well even when only 25% of the dataset is used for training, with the notable exception of the fisherc classifier. Classification accuracy increases with increasing training ratio, but when too much of the dataset is used for training (90% here) not enough data is available for validation and the variance of the classification accuracy increases. A more in-depth look at this phenomenon comes from the confusion matrices, shown in Figure 6.14. For future analysis we choose a train/test ratio of 0.6 to ensure we have enough data for validation, with the caveat that our classification performance could be a few points higher if we used a higher training ratio.

Table 6.8: Increasing the amount of data used for training increases the classification accuracy, but reduces the amount of available data for validation. As too much of the available data is used for training the classifier becomes overtrained and the variance of the accuracy measurement increases. Data is from the 100-900 approaching vehicle dataset with classification into three classes (c, s, vtb).

<table>
<thead>
<tr>
<th>Train %</th>
<th>nmsc</th>
<th>perlc</th>
<th>ldc</th>
<th>fisherc</th>
<th>udc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.82 ± 0.06</td>
<td>0.77 ± 0.06</td>
<td>0.52 ± 0.08</td>
<td>0.43 ± 0.09</td>
<td>0.72 ± 0.08</td>
<td>0.65</td>
</tr>
<tr>
<td>0.5</td>
<td>0.89 ± 0.05</td>
<td>0.85 ± 0.04</td>
<td>0.69 ± 0.07</td>
<td>0.74 ± 0.08</td>
<td>0.80 ± 0.05</td>
<td>0.79</td>
</tr>
<tr>
<td>0.6</td>
<td>0.89 ± 0.03</td>
<td>0.84 ± 0.06</td>
<td>0.76 ± 0.06</td>
<td>0.75 ± 0.06</td>
<td>0.82 ± 0.06</td>
<td>0.81</td>
</tr>
<tr>
<td>0.7</td>
<td>0.90 ± 0.06</td>
<td>0.87 ± 0.05</td>
<td>0.81 ± 0.06</td>
<td>0.79 ± 0.06</td>
<td>0.83 ± 0.06</td>
<td>0.84</td>
</tr>
<tr>
<td>0.8</td>
<td>0.91 ± 0.06</td>
<td>0.89 ± 0.06</td>
<td>0.79 ± 0.11</td>
<td>0.80 ± 0.10</td>
<td>0.82 ± 0.07</td>
<td>0.84</td>
</tr>
<tr>
<td>0.9</td>
<td>0.91 ± 0.08</td>
<td>0.86 ± 0.12</td>
<td>0.82 ± 0.13</td>
<td>0.82 ± 0.12</td>
<td>0.88 ± 0.10</td>
<td>0.86</td>
</tr>
</tbody>
</table>
Figure 6.14: This example of the classification of 98 instances pulled from the 100-900 dataset shows how increasing the training ratio first improves classification performance and then increases variance due to overtraining and a lack of data to validate the classifier. The mean confusion matrix is shown for the fisher c classifier at 25% training data (top), 60% train (middle), and 90% train (bottom).
Repeatability of classification results

Since our code creates a dataset for classification by randomly selecting observations from a given class from among all the total possibilities, we would expect some variability between these separate datasets. Table 6.9 shows the classification results for three datasets compiled from all available data from the 100-900 and 250-comb overall datasets.

Table 6.9: Classification of datasets whose instances are selected at random from the larger dataset containing all possible observations shows repeatable results for both 100-900 and 250-comb data. The overall lower performance of the 250-comb datasets is likely due to the greater variety in the observations present in this dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Classifier</th>
<th>nmsc</th>
<th>perl</th>
<th>ldc</th>
<th>fisherc</th>
<th>udc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-900:</td>
<td>a</td>
<td>0.81 ± 0.04</td>
<td>0.84 ± 0.05</td>
<td>0.78 ± 0.05</td>
<td>0.78 ± 0.06</td>
<td>0.76 ± 0.07</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0.84 ± 0.06</td>
<td>0.76 ± 0.09</td>
<td>0.69 ± 0.08</td>
<td>0.74 ± 0.08</td>
<td>0.71 ± 0.08</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>0.86 ± 0.05</td>
<td>0.84 ± 0.05</td>
<td>0.79 ± 0.07</td>
<td>0.77 ± 0.06</td>
<td>0.72 ± 0.05</td>
<td>0.77</td>
</tr>
<tr>
<td>250-comb:</td>
<td>a</td>
<td>0.60 ± 0.06</td>
<td>0.55 ± 0.06</td>
<td>0.56 ± 0.06</td>
<td>0.56 ± 0.03</td>
<td>0.53 ± 0.04</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0.58 ± 0.05</td>
<td>0.55 ± 0.06</td>
<td>0.56 ± 0.06</td>
<td>0.54 ± 0.03</td>
<td>0.56 ± 0.05</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>0.52 ± 0.06</td>
<td>0.45 ± 0.05</td>
<td>0.51 ± 0.06</td>
<td>0.50 ± 0.05</td>
<td>0.50 ± 0.04</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Classification performance is similar among both the 100-900 and 250-comb datasets. The 250-comb dataset has an overall lower classification performance, likely due to the greater variety of observations present in the dataset. The 250-comb data is a combination of 250-500 and 250-1000 data, created with the assumption that the time between chirps is less important than the length of the chirp. A comparison of classification performance of all the 250 ms chirps, shared in Table 6.10 and Figure 6.15 calls this into question.

While it is possible that this particular 250-comb dataset that was randomly selected from the largest and most diverse dataset just suffered from bad luck, there is clearly a difference in classification performance between the 250-500/250-1000
Figure 6.15: A comparison of the confusion matrices (perlc classifier) of the 250-500 (top), 250-1000 (middle), and 250-comb (bottom) datasets shows that the non-combined datasets have a much higher classification performance than the 250-comb dataset. Both the 250-500 and 250-1000 datasets are small, with 17 and 19 total instances respectively, compared to the 215 total instances of the 250-comb dataset. The classification performance of the 250-1000 dataset is almost perfect.
Table 6.10: A comparison of classification performance between the 250- datasets shows that the spacing between chirps is important in defining our transmitted signal.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>nmsc</th>
<th>perlc</th>
<th>ldc</th>
<th>fisher</th>
<th>udc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>250-500</td>
<td>0.92 ± 0.09</td>
<td>0.85 ± 0.12</td>
<td>0.72 ± 0.16</td>
<td>0.75 ± 0.14</td>
<td>0.70 ± 0.15</td>
<td>0.79</td>
</tr>
<tr>
<td>250-1000</td>
<td>0.99 ± 0.03</td>
<td>0.98 ± 0.05</td>
<td>0.56 ± 0.22</td>
<td>0.68 ± 0.19</td>
<td>0.97 ± 0.07</td>
<td>0.84</td>
</tr>
<tr>
<td>250-comb: a</td>
<td>0.60 ± 0.06</td>
<td>0.55 ± 0.06</td>
<td>0.56 ± 0.06</td>
<td>0.56 ± 0.03</td>
<td>0.53 ± 0.04</td>
<td>0.56</td>
</tr>
</tbody>
</table>

datasets and the combined 250-comb dataset. This leads us to believe that the entire transmitted signal, including the space between chirps, is important in defining a signal, rather than just the chirp length.

That said, both the 250-500 and 250-1000 datasets exhibit good classification performance, albeit with large variances. These large variances are caused by the relatively small number of useful observations in each dataset. Figure 6.16 shows an example of how these variances play out and why the confusion matrices are so important to understanding the classification results. Here, both the ldc and udc classifiers have an average overall correct rate of around 70%, but the udc classifier has difficulty correctly classifying the van/truck/bus class.

This example also illustrates the importance of having large datasets to create training and testing sets with a sufficient number of observations for each class. Both the 250-500 and 250-1000 datasets used here have fewer than 10 instances per class, meaning that even at a 60% training percentage, the classification can only be tested on a few instances.

We are forced to use these small datasets in this situation because our automated detection routing has a good deal of difficulty locating peaks from the 250- signals. The detection rate of this 250-500 dataset is 26% and the rate for the 250-1000 dataset is 29%. This is compared to a detection rate of 89% for the 100-900 signal.
Figure 6.16: A comparison of the confusion matrices from the ldc (top) and udc (bottom) classifiers on data from the 250-500 dataset highlights the importance of using the extra information present in the confusion matrix. Both classifiers have a mean overall correct rate of approximately 70%, but the udc classifier has much more difficult time classifying vans/trucks/buses into the correct class.
For this reason, and reasons discussed earlier, the 100-900 signal remains our preferred transmission signal.

**Head-on vs. oblique reflections**

Another useful comparison is between data acquired 'head-on' and at a slight angle. Results from stationary vehicles discussed in Section 5.3.2 confirm that the recorded signal contains reflected pulses, and Table 6.11 shows that both datasets have a similar classification performance. With an average correct classification rate of 58% for both, this 250-comb data isn't an ideal dataset for reasons that we discussed above, but was the only dataset containing observations from both orientations.

Table 6.11: A comparison of 250-comb data acquired in a 'head-on' orientation and data collected at a slight angle shows that both orientations have a similar classification performance.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>nmsc</th>
<th>peric</th>
<th>ldc</th>
<th>fisherc</th>
<th>udc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head-on</td>
<td>0.64 ± 0.08</td>
<td>0.56 ± 0.11</td>
<td>0.54 ± 0.11</td>
<td>0.52 ± 0.09</td>
<td>0.65 ± 0.12</td>
<td>0.58</td>
</tr>
<tr>
<td>Oblique</td>
<td>0.63 ± 0.09</td>
<td>0.56 ± 0.07</td>
<td>0.57 ± 0.10</td>
<td>0.55 ± 0.12</td>
<td>0.58 ± 0.08</td>
<td>0.58</td>
</tr>
</tbody>
</table>

**Comparison of all input signals and classification into five classes**

Finally, we make an overall comparison between the different incident signals for the three-class problem, and make an attempt at classifying the data from one dataset into the five original, non-grouped classes.

Table 6.12 shows the results from these comparisons. Even though our reflection detection algorithm has difficulties with both the 5-750 and 10-750 datasets (as well as with the 250- datasets as discussed earlier) and can only detect the reflection in 25% of the observations, we get good classification performance. The ldc and fisherc
classifiers give anomalously low mean overall classification performance rates with high variance. Removing these classifiers, we can calculate an average performance for the remaining three classifiers, shown in the last column of Table 6.12. With mean overall classification rates ranging from 82% to 98% we can’t say much about one signal being preferred to another, except that our algorithms are able to detect the reflections in the 100-900 signal best.

Table 6.12: A comparison of datasets from all of the transmitted signal types shows very good classification performance for all classifiers except ldc and fisherc. Removing these classifiers gives the average in the last column (avg2). The 100-900 and 250-datasets have been discussed previously in more detail and are included here for completeness. The final row shows the lone five-class classification attempt, with only 7 instances per class.

<table>
<thead>
<tr>
<th>Signal</th>
<th>nmsc</th>
<th>perc</th>
<th>ldc</th>
<th>fisherc</th>
<th>udc</th>
<th>Avg</th>
<th>Avg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-750</td>
<td>0.98 ± 0.05</td>
<td>0.99 ± 0.02</td>
<td>0.53 ± 0.19</td>
<td>0.54 ± 0.14</td>
<td>0.96 ± 0.07</td>
<td>0.80</td>
<td>0.98</td>
</tr>
<tr>
<td>10-750</td>
<td>0.96 ± 0.06</td>
<td>0.94 ± 0.04</td>
<td>0.61 ± 0.13</td>
<td>0.50 ± 0.17</td>
<td>0.88 ± 0.09</td>
<td>0.78</td>
<td>0.93</td>
</tr>
<tr>
<td>100-900</td>
<td>0.89 ± 0.06</td>
<td>0.86 ± 0.05</td>
<td>0.77 ± 0.07</td>
<td>0.80 ± 0.09</td>
<td>0.85 ± 0.06</td>
<td>0.83</td>
<td>0.87</td>
</tr>
<tr>
<td>250-500</td>
<td>0.92 ± 0.09</td>
<td>0.85 ± 0.12</td>
<td>0.72 ± 0.16</td>
<td>0.75 ± 0.14</td>
<td>0.70 ± 0.15</td>
<td>0.79</td>
<td>0.82</td>
</tr>
<tr>
<td>250-1000</td>
<td>0.99 ± 0.03</td>
<td>0.98 ± 0.05</td>
<td>0.56 ± 0.22</td>
<td>0.68 ± 0.19</td>
<td>0.97 ± 0.07</td>
<td>0.84</td>
<td>0.98</td>
</tr>
<tr>
<td>100-900 5C</td>
<td>0.94 ± 0.09</td>
<td>0.92 ± 0.08</td>
<td>0.47 ± 0.14</td>
<td>0.45 ± 0.19</td>
<td>0.82 ± 0.15</td>
<td>0.72</td>
<td>0.89</td>
</tr>
</tbody>
</table>

Even with the severely limited data available for classification into five classes (only 7 instances per class), we surprisingly find good classification performance, with an average classification rate of 89%. The best (nmsc at 94%) and worst (udc at 82%) classifiers for this data is shown in Figure 6.17.

In conclusion, we have shown that oncoming vehicles can be classified with a high amount of accuracy, and at useful distances, using only reflected acoustic pulses. Finding and aligning these reflected signals is a nontrivial, vital step in the process, but one that can be successfully automated, especially if the transmitted signal is optimized to the application. Useful feature vectors that differentiate between vehicles of different
Figure 6.17: Classification of 100-900 data into five classes is only possible with a limited dataset of 7 instances per class, but manages mean classification performance rates of 82% for the udc classifier (top) and 94% for the nmsc classifier (bottom).
classes can be formed by using the Dynamic Wavelet Fingerprint to create alternative time-frequency representations of the reflected signal, and intelligent feature selection algorithms create information-dense representations of our data that allow for very fast and accurate classification.

We have found that a 100-900 linear chirp transmitted signal is best optimized for this particular problem. The signal contains enough energy to propagate long distances, while remaining compact enough in time to allow easy automatic detection. With this signal we can consistently attain correct overall classification rates upwards of 85% at distances of 50 m.
Chapter 7

Simulations of scattering from a nonlinear acoustic beam

A number of questions arose from our initial pattern classification analysis of acoustic backscatter reflections from vehicles, discussed in Chapters 5 and 6. Here we use numerical simulations to gain further understanding of the propagation of the acoustic signal and its interaction with scattering targets to answer these questions.

In particular, ensuring that the acoustic signal is incident on the target and not other objects in the environment is a direct way to reduce clutter in the measured backscatter signal. We use numerical solutions to the nonlinear KZK equation to model the signal propagating from the acoustic parametric array to the target vehicle. These simulations show how the acoustic parametric array uses the nonlinearity of air to create an audible acoustic beam and allows us to visualize the directionality of the acoustic signal as it propagates.

A closer look at the interaction of the acoustic signal with real-world targets comes from three-dimensional acoustic finite integration (AFIT) simulations. From these simulations we can visualize the time evolution of the three-dimensional scattered...
pressure field as an acoustic beam interacts with various objects. This will allow us to observe exactly how the energy is scattered from different vehicle models and confirm our expectations that the measured backscattered signals contain reflections from our target of interest.

In addition, we can look at the time evolution of the pressure field at a specific spatial coordinate at a high sampling rate, analogous to experimental microphone measurements. These simulated measurements allow us to investigate the effect of varying duration incident signals on the backscattered signal, as well as confirm that the structure of our coded signals remains intact throughout the scattering process. Processing this simulated microphone data through the DWFP feature extraction algorithms provides insight into a complicated and abstract signal processing system.

Results from these simulations allow us to optimize different steps of the data collection and pattern classification process and can lead to increased classification accuracy for existing classes and future expansion of the classification to include more vehicle subtypes.

7.1 The acoustic parametric array

As described in Chapter 5, we are investigating the use of acoustic echolocation sensors for mobile robotics applications, in particular, by attempting to detect and classify oncoming vehicles using only this acoustic backscatter signal. We are most interested in the applicability of acoustic echolocation sensors as medium- to long-range sensors, detecting and classifying objects at distances exceeding 50 m. Such long propagation distances limit the upper frequency of the acoustic signal, since the absorption of sound in air is proportional to the frequency squared, requiring an acoustic signal in the audible range.
Another important aspect of the acoustic echolocation sensor is the directionality of the signal. A directional signal will allow us to focus most of the sound energy at the target in question, increasing the amplitude of the backscattered signal and the probability that the signal contains useful information about the target. This is especially important in unstructured environments with many objects in the environment that will add clutter to the backscattered signal.

In general, the directivity of a speaker depends on the ratio of the wavelength of sound produced to the physical size of the speaker. Sound in the audible range will have wavelengths between 17 m and 17 mm, calculated as $\lambda = \frac{\xi}{f}$ for frequency $f$ and sound speed $c = 343 \text{ m/s}$ in air. This explains why a normal bookshelf-sized stereo speaker will produce directional high frequency sound, since the wavelength much less than the size of the speaker, but non-directional room-filling low-frequency sound. Creating highly directional, low-frequency sound would require an impractically large array. Even the 1-4 kHz acoustic signal that we have used in our experimental measurements contains wavelengths of 8.5-34 cm, requiring a loudspeaker array several meters large in order to create a highly directional signal.

An alternative way to create highly-directional low-frequency sound is the acoustic parametric array [130, 131]. Described further in Chapter 4, this device exploits the nonlinearity of air to create a highly directional beam of sound, even at low-frequencies. Physically, the device is comprised of many small ultrasound transducers that simultaneously produce waves over a range of ultrasonic frequencies. As the large-amplitude ultrasonic signals propagate through a nonlinear medium, nonlinear effects create signals at the sum and difference frequencies. Since attenuation is proportional to the square of frequency, at large propagation distances only the difference (audible) frequency remains. This process is illustrated in Figure 7.1.

Because the parametric array is only emitting ultrasonic signals, it can be fairly
Figure 7.1: Results from the nonlinear KZK simulation shows the operation of an acoustic parametric array. The array generates two frequencies in the ultrasonic range (shaded region) at 48 and 52 kHz. As these signals propagate they self-demodulate due to the nonlinearity of air, creating a signal at the 4 kHz difference frequency and the 100 kHz sum frequency (not shown). As the signal continues to propagate away from the array, the higher frequencies attenuate faster and after 6 m only the low-frequency difference signal remains. This low frequency signal will propagate long distances in a tightly-controlled beam.
small yet create directional beams of low frequency sound. These devices, examples of which are given in Figure 7.2 are commercially available and can connect to mp3 players or computers using a standard 1/8” phone jack, though high fidelity reproduction of music is difficult due to the narrow bandwidth inherent in the design of the devices [98, 132].

Figure 7.2: A number of parametric arrays are available commercially: the Sennheiser Audiobeam (left), Audio Spotlight (upper right), and Kickstarted Soundlazer (bottom right) all use an array of ultrasonic transducers to create spatially well-controlled beams of low-frequency sound.

Following we give a mathematical description of the nonlinear acoustic beam and discuss efficient numerical simulations that will allow us to visualize the behavior of the beam propagation over real, physical distances.
7.1.1 Modeling nonlinear acoustic wave propagation

Acoustic waves are mathematically described by the wave equation, a derivation of which can be found in any number of introductory acoustics texts [133, 30]. Briefly, this derivation requires the conservation of mass, conservation of momentum, and a state equation

\[ \frac{\partial p}{\partial t} + \nabla \cdot (\rho v) = 0 \]  
\[ \rho \left( \frac{\partial v}{\partial t} + (v \cdot \nabla)v \right) = -\nabla p \]  
\[ \nabla p = c^2 \nabla \rho \]  

with variables of pressure $p$, particle velocity $v$, material density $\rho$, and sound speed $c$. A statement of the conservation of mass, (7.1) is also referred to as the continuity equation and (7.2) are the Navier-Stokes equations. Combining these equations and linearizing leads to the linear wave equation

\[ \nabla^2 p = \frac{1}{c_0^2} \frac{\partial^2 p}{\partial t^2}, \]  

described in terms of small amplitude sound speed $c_0$. This linear wave equation has known analytic solutions and provides a good mathematical description of low-amplitude signals in a homogeneous medium. In Section 7.2 we will further discuss implementation of a finite difference solution of the linear wave equation to study the scattering of acoustic waves from real-world objects.

However, the linear wave equation does not adequately model sound propagation from the acoustic parametric array, which relies on nonlinear effects as a mechanism of sound production. To describe this nonlinear sound propagation, we start with the same constituent equations, but don’t linearize. Including nonlinear terms up to the
second order leads to the Westervelt equation

\[
\nabla^2 p = \frac{1}{c_0^2} \frac{\partial^2 p}{\partial t^2} - \frac{\delta}{c_0^3} \frac{\partial^3 p}{\partial t^3} - \frac{\beta}{\rho_0 c_0^4} \frac{\partial^2 p^2}{\partial t^2}.
\]

(7.5)

Here, \( \delta \) is the sound diffusivity, described in terms of the shear viscosity \( \mu \), bulk viscosity \( \mu_B \), thermal conductivity \( k \), and specific heats at constant volume and pressure \( c_v \) and \( c_p \)

\[
\delta = \frac{1}{\rho_0} \left( \frac{4}{3} \mu + \mu_B \right) + \frac{k}{\rho_0} \left( \frac{1}{c_v} - \frac{1}{c_p} \right).
\]

(7.6)

The coefficient of nonlinearity, \( \beta \), is described in terms of the measured nonlinearity parameter \( B/A \) [134]

\[
\beta = 1 + \frac{B}{2A}.
\]

(7.7)

Higher values of \( \beta \) (and likewise, \( B/A \)) correspond to greater nonlinear effect. The value of \( B/A \) for air at 20° C is 0.4, compared to a value of 5 in water at the same temperature. This is why propagation of sound in air can be explained as a linear phenomenon in many cases. On the other hand, body fat has a \( B/A \) value of 9.9, requiring a nonlinear wave equation to accurately describe the wave propagation in body tissues [135, 136].

For plane progressive waves, the Westervelt equation (7.5) can be rewritten in a one-dimensional form as the Burgers equation

\[
\frac{\partial p}{\partial z} = \frac{\beta p}{\rho_0 c_0^3} \frac{\partial p}{\partial \tau} + \frac{\delta}{2c_0^3} \frac{\partial^2 p}{\partial \tau^2}.
\]

(7.8)

This is the simplest model for progressive plane waves to include nonlinearity and losses, and uses retarded time \( \tau \) defined as

\[
\tau = t - \frac{z}{c_0}.
\]

(7.9)
A more general description of nonlinear wave propagation comes from the Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation

$$\frac{\partial^2 p}{\partial z \partial \tau} = \frac{c_0}{2} \nabla_1^2 p + \frac{\delta}{2 c_0^2} \frac{\partial^3 p}{\partial \tau^3} + \frac{\beta}{2 \rho_0 c_0^3} \frac{\partial^2 p^2}{\partial \tau^2},$$

(7.10)

which describes the propagation of a directional sound beam along spatial dimension $z$ while accounting for diffraction, thermoviscous absorption ($\delta$), and nonlinearity ($\beta$). For these axisymmetric beams, $\nabla_1^2 p = \frac{\partial^2}{\partial \tau^2} + \frac{1}{r} \frac{\partial}{\partial r}$, while for the case of plane waves, $\nabla_1^2 p = 0$ and the KZK equation reduces to the Burgers equation.

The KZK equation is a parabolic approximation which makes the assumption that effects due to diffraction are much larger than those due to nonlinearity, i.e., that transverse changes to the wave are much larger than axial changes. This approximation introduces errors at more than $20^\circ$ from the axis and within several radii of source. Alternative models exist to more accurately model wave behavior very near the source, but our current application does not require us to know the pressure nearfield.

Exact analytical solutions of the KZK equation are not available but approximate analytical solutions are realizable by using the quasilinear solution, which involves solving the linear problem first and using that result as a source for the nonlinear solution. A few important results can be gleaned from these solutions. First, audible level is proportional to the square of both the ultrasonic level and the modulation envelope and directly proportional to the transducer area. In addition, a $+12$ dB/octave equalization curve means that generating sound at low frequencies require more ultrasound energy [132].
7.1.2 Numerical solutions of the KZK equation

Due to the intractability of exact analytic solutions of the KZK equation, we will instead focus on numerical solutions. The first numerical solutions of the KZK equation were Lee & Hamilton's Texas KZK code [137, 138]. This is a one-dimensional axisymmetric finite difference, time domain (FDTD) code to simulate propagation of a nonlinear sound beam. Normally a one-dimensional simulation wouldn't be directly comparable to real-life situations, but given the parabolic approximation inherent in the KZK equation, it works well in this case.

Our solutions use an improved version of the Texas KZK code that runs in Java through a Matlab interface [139]. This code adds absorbing boundary conditions to the simulation which allows us to greatly reduce our simulation space. Though not a parallelized code, use of the SciClone computing cluster [140] allows for increased runtimes and simulations of real-world spaces and situations.

Figures 7.3, 7.4, and 7.5 show the result of KZK simulations for a parametric array the same size as the Sennheiser Audiobeam with various levels of physical focus. Since the array has a center frequency of 40 kHz, two sinusoidal signals at 35 and 45 kHz are used as input to produce an audible signal at 10 kHz. These simulations show us the effect of physical focus, diffraction, and nonlinearity on the beam propagation.

Figure 7.6 shows the results of the KZK simulation for the same sized parametric array using an input signal that approximates 0-10 kHz white noise. This is accomplished by randomly selecting several hundred frequencies in the range of 35-45 kHz which demodulate as the signal propagates to create an audible signal. The pressure fields of both signals is shown in Figure 7.7.

One of the benefits of the KZK simulations is that they produce real-world sized beam patterns. These patterns can be superimposed on pictures of physical situations to illustrate the spatial distribution of sound from different sized parametric arrays.
Figure 7.3: Changing variables in the KZK simulation allow us to see how each part of the KZK equation affects the final beam pattern. Here we are looking at a side view of the beam, with the source at zero and propagation to the left up to a distance of 12 m. The color scale represents maximum pressure levels over the entire frequency range of the simulation (set to 120dB at the face of the transducer). As a paraxial simulation, the beam pattern is symmetric about the radial distance of zero. Results are shown for an arrays with physical focus ranging from 1 m to 20 m (which approximates an unfocused parametric array). This simulation shows the results of the full KZK simulation, including effects due to absorption, diffraction, and nonlinearity.
Figure 7.4: Changing variables in the KZK simulation allow us to see how each part of the KZK equation affects the final beam pattern. Here we are looking at a side view of the beam, with the source at zero and propagation to the left up to a distance of 12 m. The color scale represents maximum pressure levels over the entire frequency range of the simulation (set to 120dB at the face of the transducer). As a paraxial simulation, the beam pattern is symmetric about the radial distance of zero. Results are shown for an arrays with physical focus ranging from 1 m to 20 m (which approximates an unfocused parametric array). This simulation shows the results of the KZK simulation excluding the effects of diffraction.
Figure 7.5: Changing variables in the KZK simulation allow us to see how each part of the KZK equation affects the final beam pattern. Here we are looking at a side view of the beam, with the source at zero and propagation to the left up to a distance of 12 m. The color scale represents maximum pressure levels over the entire frequency range of the simulation (set to 120 dB at the face of the transducer). As a paraxial simulation, the beam pattern is symmetric about the radial distance of zero. Results are shown for an arrays with physical focus ranging from 1 m to 20 m (which approximates an unfocused parametric array). This simulation shows the results of the KZK simulation excluding the effects of nonlinearity. Because there is no nonlinear self-demodulation, the ultrasonic signals simply attenuate.
Figure 7.6: Changing variables in the KZK simulation allow us to see how each part of the KZK equation affects the final beam pattern. Here we are looking at a side view of the beam, with the source at zero and propagation to the left up to a distance of 12 m. The color scale represents maximum pressure levels over the entire frequency range of the simulation (set to 120dB at the face of the transducer). As a paraxial simulation, the beam pattern is symmetric about the radial distance of zero. Results are shown for an arrays with physical focus ranging from 1 m to 20 m (which approximates an unfocused parametric array). This simulation shows the results of the full KZK simulation, including effects due to absorption, diffraction, and nonlinearity, using a more realistic source that will produce noise in the 0-10 kHz audible range.
Figure 7.7: The simplest pressure field used as an input signal for the KZK simulation consists of two sinusoids at 35 and 45 kHz, with an array center frequency of 40 kHz. This signal (top) will produce a beam with an audible frequency at 10 kHz. A more realistic signal (bottom) is composed of individual signals at several hundred frequencies randomly distributed between 35 and 45 kHz. This will produce an audible signal that approximates 0-10 kHz white noise.
Along with easy investigation of other physical parameters, such as an array's physical focus, this provides a valuable tool to design a parametric array that creates the best beam coverage for a particular application.

As an example, Figure 7.8 shows simulation results overlaying the beam patterns of an acoustic parametric array on a physical area. From this representation we can see that the parametric array creates a tight beam of low-frequency sound that propagates long distances. This tight spatial control is what allows the acoustic echolocation to work well, since the majority of the sound energy is interrogating our target. This both increases the amplitude of and reduces the amount of clutter in the backscattered signal, improving the performance of automated detection and classification tasks. As an added benefit, the directionality of the sound beam reduces the amount of noise pollution in the surrounding environment and allows the use of audible acoustic signals at low sound pressure levels to further reduce human annoyance.

When viewing the results from these simulations it's important to look at the frequency ranges that are of concern. As an example, Figure 7.9 shows the beam pattern for a simulation from an array the same size as the Audio Spotlight using an input signal of 0-10 kHz white noise. The simulation has a maximum frequency of 140 dB at the face of the transducer, but when a filter is applied to view the frequencies only in the audible range, we find much lower sound pressure levels and the spatial extent of the beam is more clearly visible.

These KZK simulations describe the propagation of the acoustic beam created by an acoustic parametric array and are a quick and accurate method to view the extent of the sound field. This allows us to change the settings on commercial devices to produce spatially well-controlled beams of low frequency sound that deliver a large amount of sound energy to interrogate our target while reducing the amount of noise pollution in and added clutter from the nearby environment. These simulations could
Figure 7.8: The pressure beam patterns from the KZK simulations of a parametric array is overlaid on a real-world physical area to show the extent of the direct sound field. Here we see that the parametric array creates a beam of low frequency sound that propagates in a tight beam along the street to interrogate oncoming vehicle targets while reducing clutter from extraneous objects in the environment.

also be used in the future to design acoustic parametric arrays that are optimized for specific tasks.

7.2 Simulations of acoustic scattering

Numerical simulations of the KZK equation allow us to visualize the nonlinear beam of sound produced by the acoustic parametric array. This gives us some idea of the extent of the beam and its interaction with the environment, allowing us to ensure that most of the sound energy is comes from interaction with the target of interest and not from interaction with extraneous objects in the environment ('clutter'). In order to gain a better understanding of the interaction of the acoustic beam with the
Figure 7.9: The beam pattern for simulations of a parametric array the same physical size as the Audio Spotlight using a realistic signal that produces 0-10 kHz white noise seems is less defined when the pressure data contains information from all frequencies (top) than when information from ultrasonic frequencies is filtered out (middle). If the results are filtered to contain only pressure data for audible frequencies between 2-10 kHz, the tight spatial control of the beam pattern is more clearly visible (bottom). Note the difference in color scale between the images, which gives the pressure level in dB.
target, we switch to a full-wave numerical simulation, the acoustic finite integration (AFIT) technique. This finite difference scheme has been successfully used in past work to understand the interaction of surface waves with flaws in solid materials and to detect hidden explosives using the acoustic parametric array [24, 139].

7.2.1 AFIT

A number of established methods exist to calculate approximate numerical solutions to partial differential equations. These methods allow the study of complex differential equations that may not have analytical solutions. One of the earliest numerical methods is the finite difference time domain method (FDTD), which approximates derivatives with algebraic differences. The FDTD method was first used to find solutions to Maxwell's Equations in the 1960s and has since been applied to a number of different situations with great success [141, 142, 143].

While many other numerical methods have since been introduced, FDTD simulations have the advantage of being conceptually simple and easy to implement. Finite element and boundary element methods allow for computations on complex geometries using non-structured grids, but at the expense of gridding the entire computation space prior to simulations. More recent pseudospectral and k-space methods allow rapid calculations and better handling of nonlinear problems, but are more complex to implement [144].

For our simulations we will be using the finite integration technique (FIT), which is different than the more common FDTD methods in that the differential equations are integrated over a control volume and these integrals are approximated rather than directly approximating the differential equations [145]. This has the main benefit of naturally leading to a more stable staggered-grid simulation space with simpler implementation of boundary conditions [146].
In particular, we use the acoustic finite integration technique (AFIT), the constituent equations of which will be discussed shortly. This AFIT code has been written and validated by our lab through comparison to experimental results and known analytic solutions [139, 147]. A similar technique has also been extended to model elastodynamic wave propagation in solids, were it is referred to as EFIT [148]. Current AFIT simulations are linear, do not account for viscosity, and only deal with rigid scatterers, but these are all reasonable approximations that allow for an understanding of three-dimensional multiple scattering for real-world spaces and scatterers.

To derive the necessary equations for AFIT, we start with the linearized conservation of mass (7.1) and Navier-Stokes (7.2) equations, generalized to include pressure and velocity source functions $M$ and $F$ respectively:

$$\frac{\partial p}{\partial t} + \rho_0 c_0^2 \nabla \cdot \mathbf{v} = M$$  \hspace{1cm} (7.11)

$$\rho_0 \frac{\partial \mathbf{v}}{\partial t} + \nabla p = F.$$  \hspace{1cm} (7.12)

A finite difference simulation would approximate these derivatives directly, but instead we integrate over a control volume, which is a cube in Cartesian space. Doing this for (7.11) yields

$$\int_V \frac{\partial p}{\partial t} \, dV = \int_V \left(-\rho_0 c_0^2 \nabla \cdot \mathbf{v} + M\right) \, dV.$$  \hspace{1cm} (7.13)

The divergence theorem allows us to convert one of the volume integrals into a surface integral

$$\int_V \frac{\partial p}{\partial t} \, dV = -\rho_0 c_0^2 \int_S \mathbf{v} \cdot dS + \int_V M \, dV.$$  \hspace{1cm} (7.14)
Likewise, this process can be repeated for (7.12), leading to

\[
\rho_0 \int_V \frac{\partial \mathbf{v}}{\partial t} \, dV = - \oint_S p \, dS + \int_V \mathbf{F} \, dV. \tag{7.15}
\]

The next step is to approximate the integrals in (7.14) and (7.15) over a cubic control volume of size $\Delta x \times \Delta x \times \Delta x$. This volume corresponds to a single grid cell in Figure 7.10. We follow notation used in [139] and [147] to describe the direction of the pressure and velocity components on the grid relative to the center of the current cell. The vector velocity is decomposed as $\mathbf{v} = v_1 \hat{x}_1 + v_2 \hat{x}_2 + v_3 \hat{x}_3$ and $v_1^+ \text{ and } v_1^-$ represents velocity in the positive and negative $x_1$ direction respectively.

Applying these approximations to (7.14) leads to an equation for the pressure

\[
\frac{\partial p}{\partial t} (\Delta x)^3 = -\rho_0 c_0^2 \left[ (v_1^+ - v_1^-) + (v_2^+ - v_2^-) + (v_3^+ - v_3^-) \right] (\Delta x)^2 + M(\Delta x)^3, \tag{7.16}
\]

while application to (7.15) leads to an equation for the velocity

\[
\rho_0 \frac{\partial \mathbf{v}}{\partial t} (\Delta x)^3 = - (p^+ - p^-) (\Delta x)^2 + \mathbf{F}(\Delta x)^3, \tag{7.17}
\]

which can be further expanded along the three spatial dimensions as

\[
\begin{align*}
\rho_0 \frac{\partial v_1}{\partial t} (\Delta x)^3 &= - (p_1^+ - p_1^-) (\Delta x)^2 + F_1(\Delta x)^3 \\
\rho_0 \frac{\partial v_2}{\partial t} (\Delta x)^3 &= - (p_2^+ - p_2^-) (\Delta x)^2 + F_2(\Delta x)^3 \\
\rho_0 \frac{\partial v_3}{\partial t} (\Delta x)^3 &= - (p_3^+ - p_3^-) (\Delta x)^2 + F_3(\Delta x)^3. \tag{7.18}
\end{align*}
\]

The final step is to use a difference to approximate the time derivatives remaining on the left hand side of (7.16) and (7.18). For the pressure we use the standard
central-difference
\[ p^{(t)} = p^{(t+\Delta t/2)} - p^{(t-\Delta t/2)} \]
\[ \Delta t \]
with time step \( \Delta t \) to arrive at
\[ p^{(t+\Delta t/2)} = p^{(t-\Delta t/2)} - \rho_0 c_0^2 \frac{\Delta t}{\Delta x} \left[ (v_1^{1+} - v_1^{1-}) + (v_2^{2+} - v_2^{2-}) + (v_3^{3+} - v_3^{3-}) \right] + M \Delta t, \] (7.20)

while for the velocity we use an integer indexed central difference
\[ u^{(t)} = u^{(t-\Delta t)} + \dot{u}^{(t-\Delta t/2)} \Delta t \]
(7.21)
with spatial step \( \Delta x \) to arrive at
\[ v_1^{(t)} = v_1^{(t-\Delta t)} - \frac{\Delta t}{\rho_0 \Delta x} - (p_1^{1+} - p_1^{1-}) + F_1 \frac{\Delta t}{\rho_0} \]
\[ v_2^{(t)} = v_2^{(t-\Delta t)} - \frac{\Delta t}{\rho_0 \Delta x} - (p_2^{2+} - p_2^{2-}) + F_2 \frac{\Delta t}{\rho_0} \]
\[ v_3^{(t)} = v_3^{(t-\Delta t)} - \frac{\Delta t}{\rho_0 \Delta x} - (p_3^{3+} - p_3^{3-}) + F_3 \frac{\Delta t}{\rho_0}. \] (7.22)

This discrete set of equations for pressure and velocity given by (7.20) and (7.22) provide updates to the staggered grid in space and time.

The size of the spatial step \( \Delta x \) (which is also the size of a single grid cell) is limited by the upper frequency in the simulation. In general, maintaining stability requires at least 6 grid points per wavelength. For our simulations we use 8 points per wavelength, so the minimum step size is \( \Delta x = ds = \frac{\lambda}{8} \).

The minimum time step is related to the spatial step size by the Courant condition, given by
\[ \Delta t = dt \leq \frac{\Delta x}{c \sqrt{3}}. \] (7.23)
Figure 7.10: The AFIT spatial grid has staggered pressure and velocity nodes. Image used with permission from [147].
7.2.2 Implementation of AFIT on the SciClone computing cluster

We are running a parallelized version of our AFIT code on the SciClone computing cluster located at William and Mary. This distributed computing platform contains 943 CPU cores with a total of 5.9TB of physical memory and 220 TB disk capacity divided among 193 compute cores, providing a theoretical peak floating point performance 21.2 TFLOP/S. Parallelization is accomplished using a one-dimensional virtual topology to pass information between neighboring nodes with the Message Passing Interface (MPI).

The main limit on the size of the AFIT computation that can run on SciClone is the amount of memory required to store the simulation space. Memory use is approximated as

\[ \frac{115 \, n_x \, n_y \, n_z}{10^9} \, \text{GB}, \]  

(7.24)

where \( n_x, n_y, \) and \( n_z \) is the total size of the simulation space in steps for each of the three Cartesian coordinates.

As discussed previously, because the AFIT simulation requires at least 6-8 points per wavelength to remain stable, the spatial step size (and therefore total number of steps for a given metric space size) depends on the highest frequency present in the simulation. In addition, the time step size is related to the spatial step size. All together, this means that simulations at higher frequencies require smaller spatial and time steps and thus require a larger sized simulation space and a longer run time. Table 7.1 illustrates this effect.

Another important consideration is the size and amount of output data, especially for very large simulation volumes. Our AFIT simulation has the capacity to output pressure data for the full simulation space at every time step, but by default saves
Table 7.1: The maximum frequency required for the AFIT scattering simulation determines the minimum step size in both time and frequency domains. Here we calculate the number of time steps required for different maximum frequencies using a conservative 8 points per wavelength. The speed of sound in air is 343 m/s, which means it will take a wave approximately 2.9 ms to propagate one meter.

<table>
<thead>
<tr>
<th>Max freq (kHz)</th>
<th>ds (m)</th>
<th>dt (μs)</th>
<th>Wave propagation per dt (m)</th>
<th>Min timesteps for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1m    5m  20m  50m</td>
</tr>
<tr>
<td>4 kHz</td>
<td>0.08575</td>
<td>0.0107</td>
<td>18</td>
<td>0.006174 165 810 3240 8100</td>
</tr>
<tr>
<td>8 kHz</td>
<td>0.042875</td>
<td>0.00536</td>
<td>9</td>
<td>0.003087 325 1620 6480 16200</td>
</tr>
<tr>
<td>20 kHz</td>
<td>0.01715</td>
<td>0.00214375</td>
<td>3.6</td>
<td>0.001235 810 4050 16200 40500</td>
</tr>
</tbody>
</table>

To further reduce the size of the output files, we have added the option to output binary files. This greatly reduces the size of the output files while also allowing for quicker post-simulation analysis and visualization. This file format can be read as a brick-of-values format by the open-source VisIt software to allow for easy three-dimensional visualizations of the wave propagation. The ability of VisIt to create visualizations directly on the distributed computing cluster is especially important for larger simulation space sizes.

We are concerned with scattering behavior in the low-frequency acoustic regime, since our incident signal has a maximum frequency content of 4kHz. At these frequencies, even large spaces are possible. Table 7.2 shows some approximations of memory and disk use for our simulations.
Table 7.2: The memory requirements of the finite-difference scattering simulation depends on the size of the simulation space. Results shown here are for our 4kHz simulations. Higher frequencies require smaller spatial and time steps, increasing the size of the simulation space. The overall output is total file size for a simulation in which the acoustic wave propagates across the entire space, with a time resolution of 1 ms, corresponding to every 55 time steps for our simulations.

<table>
<thead>
<tr>
<th>Space (m)</th>
<th>Space (steps)</th>
<th>Memory</th>
<th>Output file size</th>
<th>Min timesteps</th>
<th>Overall output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 1 x 1</td>
<td>94 x 94 x 94</td>
<td>95 MB</td>
<td>830 KB</td>
<td>160</td>
<td>2.5 MB</td>
</tr>
<tr>
<td>5 x 5 x 5</td>
<td>467 x 467 x 467</td>
<td>11.7 GB</td>
<td>102 MB</td>
<td>810</td>
<td>1.4 GB</td>
</tr>
<tr>
<td>10 x 10 x 10</td>
<td>935 x 935 x 935</td>
<td>94 GB</td>
<td>817 MB</td>
<td>1620</td>
<td>24 GB</td>
</tr>
<tr>
<td>5 x 5 x 50</td>
<td>467 x 467 x 4673</td>
<td>117 GB</td>
<td>1.02 GB</td>
<td>8100</td>
<td>150 GB</td>
</tr>
<tr>
<td>10 x 10 x 10</td>
<td>935 x 935 x 4673</td>
<td>470 GB</td>
<td>4.09 GB</td>
<td>8100</td>
<td>601 GB</td>
</tr>
</tbody>
</table>

7.3 Simulations of acoustic scattering from vehicles

To study the acoustic scattering from vehicles we first create some models of real-world scattering objects and import them into our computational space. Visualizations of the scattered pressure field will allow us to study how a scatterer's shape affects the backscattered reflection, providing useful specific information that will add to our intuitive knowledge of scattering behavior.

We can also use the pressure field visualizations to investigate the effects of the duration of the incident acoustic signal, although computational restrictions on the size of the simulation space limit the signal length. An alternative visualization of the time evolution of pressure at a specific spatial coordinate can be calculated in a manner that overcomes this restriction, allowing us to study reflections from any incident signal. These simulated pressure signals are similar to the experimentally recorded microphone measurements and can also be used to investigate how the structure of the incident acoustic signal changes as it reflects from a scatterer.

Finally, we can use the simulated microphone data to further understand the
DWFP feature extraction and feature selection steps of the pattern classification routine described for our experimental data in Chapter 6. While the data is still complicated enough that no one feature will highlight the differences between reflections from different vehicle models, this is a more manageable dataset than the experimental data.

### 7.3.1 Creating vehicle scatterers

To create vehicle models of appropriate detail, we must consider the type of scattering we can expect. The particular type of scattering is determined by the relation of the wavelength of the incident wave to the size of the scatterer. If the wavelength is much less than the size of the scatterer, the laws of geometrical optics (for example, Snell’s law) describe the interaction. Similarly, if the wavelength of the incident signal is much greater than the size of the scatterer (no less than 10 times greater), Rayleigh scattering dominates and the exact shape of the scatterer is of little importance. Analytical solutions exist that describe this scattering behavior, with the main result that scattering intensity is inversely proportional to the wavelength as \( I \propto \lambda^{-4} \).

The most interesting scattering behavior occurs when the wavelength of the incident signal is roughly the same as the size of the scatterer, referred to as Mie scattering. Closed form solutions exist for Mie scattering from simple shapes such as spheroids, but there is no general solution for arbitrary shapes. Solutions to such scattering problems are most often solved using numerical solutions, as we are doing here. Acoustic and electromagnetic scattering has a very rich literature reaching back to the mid-1800s and cannot be discussed in detail here except as it pertains to understanding the scattering from vehicles [149]. Further information may be found in standard texts [150, 151].

We can use this knowledge of scattering behavior to understand what level of
detail needs to be included in our vehicle models. Creating models that are too
detailed will unnecessarily complicate the numerical simulation without adding any
useful information on scattering behavior, and possibly even producing artifacts in
the simulation results. On the other hand, creating models that are not detailed
enough will not give us any useful information that will allow us to differentiate
vehicle models based on the backscattered pressure field. The 1-4 kHz linear acoustic
chirp signal used in the experimental measurements has wavelengths between 8-35
cm, and structures approximately this size will interact most strongly with our signal.
However, the candidates for small-sized features to include in our models are grills,
headlights, and bumpers, all of which add more detail than necessary. Therefore, even
though our 4 kHz simulations has the ability to accurately describe these small scale
features with a spatial step size of 0.0107 m, our vehicle models will consist of angled
planes. Further refinements to these models could include side view mirrors and
curved surfaces, but care must be taken to keep the models generic enough to describe
all possible variants of vehicles in a specific class. Another option is to create models
for multiple specific vehicles within a larger class to compare inter-class scattering
differences.

A number of publications discuss the topic of radar scattering from vehicles, but
mostly through experimental measurements of radar cross section [152, 112]. In
addition, scattering from an acoustic wave is much simpler than scattering from an
electromagnetic wave since the scattering field is scalar. Likewise, while windshield
glass may be transparent to radar, leading to a greater scattering contribution from
structures in the interior cabin of the vehicle, an incident acoustic signal will treat
the glass as a rigid scatterer.

We expect that the greatest contribution to the backscattered signal will come
from vertical surfaces that are parallel or near-parallel to the plane of the acoustic
parametric array. For vehicles, this would include the front grill and windshield for all vehicles, as well as the engine cover (hood) for some vehicles. Further, we expect that the backscattered signal from the bus and box truck will be of the highest amplitude due to the large size of these scattering surfaces.

Previous AFIT simulations have used scatterers that are built from simple shapes or created from sliced three-dimensional images [139, 147]. Building complex objects from simple shapes directly in the C++ AFIT code can lead to gaps between objects and other unwanted behaviors. Using sliced three-dimensional images is a better solution that allows for fairly straightforward rotation of the scatterer to study scattering at an angle, but is an unnecessarily added step.

Instead, for our work we will directly import three-dimensional geometries into the simulation space. We have chosen STL (STereo Lithography) files from a number of common 3D file formats for their relative simplicity and ease of use. For our purposes, only the location of the vertices and connecting faces is important, and we can easily ignore additional color, lighting, or viewpoint information. STL files contain this information in a human-readable ASCII format, and data from other three-dimensional file formats can be easily exported to this format\(^1\). With the rise of 3D printers, STL files are now prevalent, and creators freely share their designs under open source licenses at sites such as Thingiverse [153]. In addition, there are many software options that can create, modify, and display STL files. A number of commercial solid-modeling tools exist, such as Autodesk Inventor and Dassault Systemes' SolidWorks [154, 155], as do several open source projects more orientated to creating three-dimensional animations, such as Blender [156].

\(^{1}\)It is also possible to create binary STL files, reducing the file size in exchange for the lack of human readability, though this is not popular.
manipulate simple three-dimensional models for use in our scattering simulations [157, 158]. Unlike typical solid modeling programs, OpenSCAD uses a text-driven interface to create a three-dimensional object that can be exported to an STL file. This type of interface allows creation of parametric models that can be easily modified. Creating the models ourselves allows us to specify the origin and dimensions to match real world vehicles at the level of detail that makes the most physical sense. The OpenSCAD parametric code used to create these vehicle models is included in Appendix 8.2.

From this parametric code we have created three-dimensional models of five different types of vehicles, shown in Figure 7.11. Models are truncated since we are only concerned about backscattering from the front. Approximations for the 11 input parameters for the model were found from dimensions of typical vehicles in each class, found in manufacturers’ and industry data. These parameters are: front hood height, back hood height, windshield height, grill depth, hood depth, windshield depth, tire offset, tire radius, tire width, body width, and overall length. Box trucks require two extra parameters - at which length (from the front of the vehicle) the box begins, the overall height of the box, and the distance the box extends past the cab in the y-direction (width) on each side of the truck\(^2\). Specific values of these parameters used to create each model are given in Table 7.3, and their physical meaning is shown in Figure 7.12. Model orientation in the three-dimensional space is given in Figure 7.13.

Rather than directly importing the STL model into the AFIT C++ simulation, we first read it into Matlab and include it as part of the simulation inputs. This allows easy positioning of the model in the real, three-dimensional simulation space. The vertices and faces from the ASCII STL file are read into Matlab and oriented in the

\(^2\)These parameters are 1500, 750, and 100 mm respectively for the model box truck used here.
Figure 7.11: Three-dimensional vehicle models for five vehicle classes have been created using a parametric model in OpenSCAD. These models are appropriately sized and saved as an STL file that can be imported into the AFIT simulation space. Since we are most concerned with scattering from the front of vehicles, models are truncated to reduce the size of the simulation space.
Figure 7.12: An eleven parameter model was used to create three-dimensional representations of vehicles.
Table 7.3: Simplified three dimensional models are created for five different types of vehicles using a parametric model with eleven parameters. The parameters are shown in relation to an actual vehicle in Figure 7.12 (all dimensions in mm).

<table>
<thead>
<tr>
<th>Veh. type</th>
<th>Model</th>
<th>fh</th>
<th>bh</th>
<th>wh</th>
<th>gd</th>
<th>hd</th>
<th>wd</th>
<th>to</th>
<th>tr</th>
<th>tw</th>
<th>bw</th>
<th>ol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>Volvo S40</td>
<td>480</td>
<td>800</td>
<td>480</td>
<td>50</td>
<td>1090</td>
<td>1100</td>
<td>550</td>
<td>320</td>
<td>235</td>
<td>1770</td>
<td>2500</td>
</tr>
<tr>
<td>SUV</td>
<td>Chevy Suburban</td>
<td>950</td>
<td>1080</td>
<td>480</td>
<td>100</td>
<td>1200</td>
<td>800</td>
<td>470</td>
<td>400</td>
<td>275</td>
<td>2000</td>
<td>2500</td>
</tr>
<tr>
<td>Van</td>
<td>Chevy Astro</td>
<td>780</td>
<td>1090</td>
<td>600</td>
<td>100</td>
<td>550</td>
<td>750</td>
<td>250</td>
<td>400</td>
<td>275</td>
<td>2150</td>
<td>2000</td>
</tr>
<tr>
<td>Truck</td>
<td>Isuzu N-series</td>
<td>1000</td>
<td>1050!</td>
<td>750</td>
<td>20</td>
<td>20</td>
<td>400</td>
<td>650</td>
<td>400</td>
<td>275</td>
<td>2050</td>
<td>2000</td>
</tr>
<tr>
<td>Bus</td>
<td>GM T6H</td>
<td>1275</td>
<td>1275</td>
<td>1530</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>1880</td>
<td>500</td>
<td>300</td>
<td>2600</td>
<td>3000</td>
</tr>
</tbody>
</table>

Figure 7.13: The model axes are aligned to allow for propagation in the positive z-direction and are entirely positive with an origin located at the bottom front of the vehicle.

The simulation space. Check steps ensure that the entire scatterer fits into the simulation space. The vertices are then converted from the real-world dimensions to dimensions in the simulation space using the step size. A Matlab routine using the ‘inpolyhedron’ function allows easy determination of which points are inside the scattering object, which are saved to a logical array and used as input to the AFIT simulation [159]. The simulation will treat these points as rigid scatterers. This process is illustrated in Figure 7.14.
Figure 7.14: A vehicle model saved as an STL file (top left) is imported into Matlab. The image is scaled to the correct grid size and the 'inpolyhedron' routine is used to find the grid points that fall inside the model (middle right). The model is then placed inside the full simulation space (bottom left), where values of one (blue dots) indicate which grid points are inside the model and values of zero (red circles) indicate points in free space. Here the process is illustrated for a simple vehicle model that was not used in calculations.
7.3.2 The effect of incident pulse length on backscattered reflection

Previous work with the EFIT simulations have been able to simulate the entire pulse interacting with flaws in materials [18]. This is due to the fact that a relatively small number of time steps are required to study the signal as it propagates to the region of interest, interacts with a flaw, and continues to propagate in multiple directions. However, while moving from elastodynamic simulations of solid materials to acoustic simulations in air reduces the complexity of the simulation (for example, fluids cannot support shear waves so there is no mode conversion in air), other factors gain importance.

While the acoustic simulations are most concerned with behavior at lower frequencies, which reduces the minimum spatial step size and increases the minimum time step size necessary for a stable simulation, the speed of sound in air (343 m/s) is also reduced when compared to the longitudinal speed of sound in materials (6420 m/s in aluminum). In addition, the size of the simulation space must be increased to relate simulations to real-world problems with length scales on the order of meters, such as our experiments of scattering from vehicles. Lastly, increased effects of air absorption require the use of longer duration signals to propagate enough energy to interrogate a distant target. Combined, these factors lead to difficulties in directly simulating the scattering behavior of acoustic signals longer than a few milliseconds in duration.

This is especially the case for our experiments of studying the scattering from vehicles. Focusing on scattering behavior below 4kHz helps reduce the size of the simulation space, leading to reasonably-sized simulation spaces as shown in Table 7.4. However, while several measurements were acquired using 5 and 10 ms duration linear chirps, the majority of the data was collected using linear chirps 100 or 250 ms
in duration. These long signals were necessary to ensure that the signal had enough energy to propagate 50 m, interrogate a target, and return 50 m to the microphone, all while competing against background noise and environmental clutter. Sophisticated signal detection and pattern classification algorithms discussed in Chapter 6 confirm that this is exactly the behavior of those signals.

Table 7.4: Using non-cubic simulation spaces reduce both the run-time memory requirements and output file size of the finite-difference scattering simulation. The space sizes shown here for our 4kHz simulations are optimized to minimize the size of the simulation space while still showing details of acoustic reflections from the vehicle. For all cases, the base of the scatterer begins at 0.25 m in the x-dimension and the scatterer is centered in the y-dimension. Having the front of the scatterer begin at 1.5 m in the z-dimension allows plenty of free space to visualize the acoustic backscatter. Overall output is the result of running the simulation for 825 time steps (enough time for the wave to propagate to the scatterer and back to the origin) with output every 25 steps, a time resolution of 0.45 ms.

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Space (m)</th>
<th>Space (steps)</th>
<th>Memory</th>
<th>Output file size</th>
<th>Overall output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>2.5 × 2.5 × 4.5</td>
<td>234 × 234 × 420</td>
<td>2.6 GB</td>
<td>23 MB</td>
<td>760 MB</td>
</tr>
<tr>
<td>SUV</td>
<td>3.5 × 3.5 × 4.5</td>
<td>328 × 328 × 420</td>
<td>5.2 GB</td>
<td>45 MB</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>Van</td>
<td>2.5 × 2.5 × 4</td>
<td>234 × 234 × 374</td>
<td>2.4 GB</td>
<td>20 MB</td>
<td>676 MB</td>
</tr>
<tr>
<td>Truck</td>
<td>3.5 × 3.5 × 4</td>
<td>328 × 328 × 374</td>
<td>4.6 GB</td>
<td>40 MB</td>
<td>1.3 GB</td>
</tr>
<tr>
<td>Bus</td>
<td>4 × 4 × 5</td>
<td>374 × 374 × 466</td>
<td>7.5 GB</td>
<td>65 MB</td>
<td>2.2 GB</td>
</tr>
</tbody>
</table>

Direct simulations of 100 and 250 ms signals is not practical on the distributed computing resources available to us for a number of reasons. In our simulation space with a time step of 1.8042 × 10⁻⁵ seconds, 100 and 250 ms signals are 5543 and 13847 samples long, respectively. Accurate modeling of such long pulses with enough clear space ahead of the scatterer to study the behavior of the backscattered reflection requires very large simulation spaces. An acoustic signal will propagate 34.3 m in 100 ms and 85.75 m in 250 ms, corresponding to 3200 and 8015 steps respectively for these simulations. Assuming that we require double this propagation distance of free space before the scatterer to observe at backscattered reflections and that
the scatterer requires and additional 5 m of space in the propagation direction, we
can calculate a minimum size of the simulation space, assuming perfect absorbing
boundary conditions and a (y,x) plane dimension of 4 m in each direction. The result,
expanded in Table 7.5, is that a 100 ms signal requires 75 m of simulation space
in the direction of propagation and a 250 ms signal requires 175 m. The memory
requirements for such space sizes are large but not unattainable at 113 GB for the
100 ms signal and 264 GB for the 250 ms signal. The larger issue is the number of
time steps required for these simulations - full propagation over 150 m (75 m there
and back for the 100 ms signal) requires 24,300 time steps while 350 m of propagation
requires 56,700 time steps. The resulting output files would be quite large as well,
such that even saving only half the space requires nearly 1 GB per file for the 100 ms
input signal and 2.3 GB per file for the 250 ms signal.

Table 7.5: Directly simulating the 100 and 250 ms duration linear chirps that were
used in experimental measurements requires a very large simulation space. Using
non-cubic simulation spaces with absorbing boundary conditions reduces both the
run-time memory requirements and output file size of the finite-difference scattering
simulation, but accurate study of the backscattering field requires a large amount
of empty space in front of the scatterer. While the memory requirements of these
computations are not unattainable, repeated iteration of the simulation to study the
scattering behavior is not practical due to the number of time steps required and the
size of the output files.

<table>
<thead>
<tr>
<th>Signal length</th>
<th>Space size</th>
<th>Memory</th>
<th>Output file size</th>
<th>Min timestep</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms steps</td>
<td>m steps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 5543</td>
<td>4 × 4 × 75</td>
<td>374 × 374 × 7010</td>
<td>113 GB 980.5 MB</td>
<td>24,300</td>
</tr>
<tr>
<td>250 13857</td>
<td>4 × 4 × 175</td>
<td>374 × 374 × 16356</td>
<td>264 GB 2.3 GB</td>
<td>56,700</td>
</tr>
</tbody>
</table>

An alternative solution is to use a delta pulse as input to the AFIT simulation.
The scattered pressure field is then basically the impulse response of the system. Since
this is a linear system, convolution of the impulse response pressure field with any
signal will provide the pressure field for that incident signal. Using this technique
to create the whole backscattered pressure field would require a convolution at each point in the volume and is not a computationally efficient procedure. However, we return to this technique later when we are looking at pressure data as a function of time at a specific spatial coordinate.

To initially evaluate the scattered pressure field from each vehicle model we will use a delta signal and a 1-4 kHz linear chirp 2.9 ms in duration (referred to as shortpulse). These signals will allow us to look for intra-class differences for a signal containing all frequencies supported by the grid (in the case of the delta pulse) and for a signal with the same frequency content of our experimental signal.

While our data consists of pressure values for each point in the three-dimensional simulation space at a given time step, displaying this data as a three-dimensional volume is often not ideal. Data in the interior of the volume, where the scattering behavior we are most interested in occurs, is obscured by data from the edges of the surface, illustrated in Figure 7.15. Decreasing the opacity of the pressure data at points on the edges of the space allows observation into the interior at the cost of a direct correlation between color values and pressure values.

A more useful three-dimensional visualization is the contour plot, which displays surfaces of constant value throughout the entire volume. These are areas of equal pressure and can be used to illustrate the scattering behavior as seen in Figure 7.16. These visualizations work well for short duration input signals, but become crowded when signals of longer duration fill the same space.

However, we have found that the best way to visualize scattering behavior is to take slices through the pseudocolor volume. This also has an advantage over volume visualizations in that it requires less graphical power to create renderings - not a problem for our modest-sized simulation spaces but an important issue in other simulations. The three-dimensional contour plot can also be sliced to create
Figure 7.15: Direct visualization of our three-dimensional pressure data at a single time step highlights information at the edges of the simulation space. The interaction between the input signal and the scatterer occurs near the middle of the simulation space, and is obscured in this visualization method, even when using opacity to peer inside the volume. At this time step, the reflection from the front of the vehicle (here, a truck) is visible at the end of the simulation space. The incident signal is a 1-4 kHz linear chirp 2.9 ms in duration.

two-dimensional contour lines.

Referring back to Figure 7.13 for orientation of our models in three-dimensional space, slices in the (x,y) plane correspond to viewing the vehicle in a head-on orientation, the (y,z) plane provides a top view, and the (x,z) plane provides a side view. The head-on view provides little useful information about the scattering behavior and will not be used. The top view, slicing along the (y,z) plane, does provide scattering
Figure 7.16: A three-dimensional contour plot allows visualization of surfaces of equal pressure at a single time. These visualizations work well for input signals of short duration. Here, the surfaces of equal pressure are shown when the shortpulse signal is first incident on the front of the truck model scatterer (top) and a short time later after the signal has been reflected from the entire truck.
information, but it is not that useful in differentiating models. Although the widths of comparison each vehicle model is slightly different, there are no other reflectors in this plane that affect the scattering behavior between different vehicle models. For this reason, only one comparison is needed of scattering in the (y,z) plane, shown for scattering from the truck model in Figure 7.17.

We are most interested in looking at the scattering behavior in slices in the (x,z) plane, where there is the most variation between vehicles. For these slices we will investigate the difference in scattering behavior using the 2.9 ms-long 1-4kHz linear chirp ('shortpulse') and delta pulse input signals. Figure 7.18 shows these input signals propagating through free space at the same speed, as expected. Both signals will be incident upon the scattering target at the same time but the longer duration shortpulse signal will interact with the target over a longer time.

The first question we would like to answer from these pressure field visualizations is how the sound energy is scattering from these vehicle models. Our experimental measurements tell us that at least some of the sound energy is reflected back toward the source, but these visualizations of the pressure field allow us to see exactly how the sound is scattered.

Comparing the scattered pressure field at several time steps will show us how the incident signal scatters from different parts of the vehicle model and how these individual reflections combine to create the larger pressure field. In addition, we can see how the frequency content of the incident signal affects the scattered field by using both the shortpulse signal, with frequency content from 1-4 kHz, and the delta pulse, with frequency content limited only by the grid spacing\(^3\). These comparisons

\(^3\)Due to direction-dependent dispersion at high frequencies, bandwidth is more limited in some grid directions than would be expected from the Nyquist sampling theorem. Here, with a sampling rate of approximately 55 kHz we would expect an upper frequency limit of 27.5 kHz but may have an effective upper frequency limit of 5.4 kHz depending on grid orientation [160].
Figure 7.17: Pseudocolor slices of the pressure field through the (y,z) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both the 2.9 ms 1-4 kHz linear chirp 'shortpulse' (top) and delta pulse (bottom) signals scattering from a truck model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. The scattering behavior is much easier to see for the delta pulse incident signal, especially the scattering from the edges that start the box of the truck (most visible in the lower right image).
Figure 7.18: Pseudocolor slices of the pressure field through the (x,z) plane along the middle of the simulation space provide a visualization of the incident signals as they propagate in free space. Here we see that both the shortpulse (top) and delta pulse (bottom) signals propagate at the same speed, as would be expected.
are shown in Figures 7.19, 7.20, 7.21, 7.22, and 7.23 for scattering from all five vehicle models (car, SUV, van, truck, and bus) using slices in the (x,z) plane.

We can make several general observations about scattering behavior from these simulation results. Most importantly we notice that the majority of the scattered sound energy propagates back toward the source location on the \((y,x)=(0,0)\) plane. This is consistent with our experimental results that measure the backscattered signal at a location very near the source location.

As expected, the main reflection comes from the vehicle’s front grill, and vehicle models with larger, vertical grills such as the bus, truck, and SUV have a larger amount of energy reflected back at the source. This makes intuitive sense that the larger cross-section presented to the incident signal, the greater the pressure amplitude of the backscattered reflection. However, the angle of the surfaces play an important role in where the reflected energy is directed. This is most obvious in the scattered field from the box truck, where we can see that sound reflected from the grill returns in the direction of the source but sound reflected from the angled windshield is directed upwards. Reflections from these other surfaces do affect the backscattered signal measured at the source location, especially for larger propagation distances where the scattered field has more time to interact with itself. Even though the total amount of energy present in a signal is an important consideration in creating a backscattered pressure field of large enough amplitude to measure at long propagation distances, long duration signals that contain more energy may not reflect those high amounts of energy back at the source.

Our models were created with the purpose of investigating intra-class differences in scattering behavior. With this in mind, Figures 7.24 and 7.25 compare the scattering behavior for the delta pulse incident signal at an early and late time step in the simulation, respectively. These images provide a closer look at the scattering behavior
Figure 7.19: Pseudocolor slices of the pressure field through the \((x,z)\) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both shortpulse (top) and delta pulse (bottom) signals scattering from a car model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. More structure is visible in reflections from the shortpulse signal due to the larger amount of energy in that signal, but both incident signals primarily reflect sound energy back toward the source. Note that the scattered field at 650 time steps (right) includes reflections from the edge of the simulation space.
Figure 7.20: Pseudocolor slices of the pressure field through the \((x,z)\) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both shortpulse (top) and delta pulse (bottom) signals scattering from a SUV model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. As for scattering from the car model, the shortpulse signal clearly shows the scattered field, but the delta pulse provides a very clear representation of the field at 350 time steps (bottom left). Due to the larger size of the front grill, even more energy is reflected back toward the source. Likewise, a more aggressively angled windshield creates greater reflections.
Figure 7.21: Pseudocolor slices of the pressure field through the (x,z) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both shortpulse (top) and delta pulse (bottom) signals scattering from a van model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. The scattered field from the van is more difficult to see in detail, but we note that the greater angle of the front grill leads to a more directional reflection.
Figure 7.22: Pseudocolor slices of the pressure field through the (x,z) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both shortpulse (top) and delta pulse (bottom) signals scattering from a truck model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. The large, flat front of the truck leads to large pressure amplitudes in the reflected signal and the reduced angle of the front grill and less aerodynamic shape reduces directional scattering effects. In these simulations we can clearly see that incidence on two differently angled surfaces causes the sound energy to be reflected in two different directions. The scattered field from the delta pulse incident signal (bottom right) also shows how these reflected signals interact as they propagate back toward the source.
Figure 7.23: Pseudocolor slices of the pressure field through the \((x,z)\) plane along the middle of the simulation space provide a visualization of the incident signals as they scatter from the vehicle model. Here we see results for both shortpulse (top) and delta pulse (bottom) signals scattering from a bus model. The images on the left show the pressure field shortly after incidence on the front of the vehicle model, while images on the right show the pressure field a short time later. Very nearly comparable to a flat wall, the bus model produces backscattered reflections at high pressure amplitudes and without any noticeable difference in directionality between the two incident signals.
in the region of interest.

### 7.3.3 Simulated microphone data from the backscattered pressure field

An alternate way of looking at the backscattered pressure field data is to view the pressure as a function of time at one location. Instead of saving the entire pressure field at a set number of time steps (typically every 25 time steps), the simulation outputs the pressure field at a particular spatial location at each time step – basically a simulated microphone measurement. This provides excellent time resolution and avoids the requirements of dealing with large output files, leaving us with a single ASCII file a few KB in size. However, since the entire pressure field must be simulated at every time point, this does not appreciably speed up the AFIT simulation.

By setting the coordinates of where this pressure data is saved, we can create a signal of pressure as a function of time $p(t)$, similar to what a microphone would record. In the last section we noticed how different input signals reflect at different angles from the vehicle models, so the location we record in the $(y,x)$ plane will certainly make a difference in the $p(t)$ signal. For simplicity, we choose the $(y,x)$ plane along the midline of the simulation space.

Choosing the location along the $z$-axis (the axis of propagation) to record $p(t)$ also requires careful planning. The input signal starts at $z=0$ and propagates through free space in the positive $z$-direction until it encounters the scatterer. Leaving enough free space before the wave encounters the scatterer is important to see the scattering behavior. In addition, we must be careful of reflections from the edges of the simulation space. As the wave propagates back towards the origin plane of $(y,x)=(0,0)$, some of the signal reflects off the boundary of the simulation space, since the absorbing
Figure 7.24: Pseudocolor slices of the pressure field through the (x,z) plane along the middle of the simulation space provide a visualization of the incident delta pulse signal as it scatters from the vehicle model. Here we compare pressure fields from all five vehicle models shortly after incidence.
Figure 7.25: Pseudocolor slices of the pressure field through the (x,z) plane along the middle of the simulation space provide a visualization of the incident delta pulse signal as it scatters from the vehicle model. Here we compare pressure fields from all five vehicle models some time after incidence when the backscattered pressure field has reached the source location.
boundary conditions only apply to the sides of the space. This is analogous to what would happen if a reflector were immediately behind the microphone, which is a fair approximation of our measurement setup.

Figure 7.26 shows the how the choice of location along the z-axis affects the simulated microphone measurement for a shortpulse reflection from a car. When the simulated pressure measurements are acquired at z=1 step, on the left edge of the boundary, we see the incident signal propagating through the measurement point and receive the backscattered reflection from the vehicle scatterer a short time later. As a comparison, saving the simulated pressure measurements at z=150 steps corresponds to a location approximately 1.6 meters inside of the simulation space. Here we see the incident signal propagating through the measurement point at a later time immediately followed by the backscattered reflection from the vehicle scatterer. The benefit of measuring at this location is that reflections from the source plane contribute less to the measured signal, but doing so requires lengthening the simulation space to include enough free space in front of the scatterer to study the scattering behavior.

A better way to calculate the backscattered pressure for long pulses

As discussed above, directly simulating long-duration incident signals requires very large simulation spaces and long runtimes. However, we are concerned with measuring the pressure as a function of time at a single spatial coordinate.

By using a delta pulse as input to the AFIT simulation, the resulting scattered pressure field represents the impulse response of the system. Point-wise convolution of this scattered field with any arbitrary signal over the entire space will provide the scattered field for that incident signal. Another benefit of this impulse response method is that the AFIT simulation needs only to be done once, and the simulation space can be kept to a minimum size, only slightly larger than the scatterer.
Figure 7.26: The simulated pressure signal is measured near the source plane at $z = 1$ step (top) and after approximately 1.6 m of propagation at $z = 150$ steps (bottom) for the shortpulse signals reflecting from a car model. While reflections from the source plane contribute less to the pressure signal measure at $z = 150$ steps, these simulations require lengthening the simulation space. In addition, reflections from the source plane mirror our experimental setup with a reflector behind the microphone.

Doing this convolution for every point in a large three-dimensional space is not practical, but will work well to calculate the pressure as a function of time at a single point. Figure 7.27 shows a comparison of a direct simulation using the shortpulse incident signal and the impulse response method convolved with the shortpulse signal.

This technique allows us to simulate the backscattered pressure data at a single point for any input signal, including long-duration signals for which direct simulations are not feasible. For example, an AFIT simulation of a linear chirp 100 ms in duration scattering from a vehicle required 25,000 time steps and took 124 wall-clock hours. A simulation of the same scenario using the delta pulse as an incident signal takes less
Figure 7.27: The simulated microphone signal is directly simulated (top, blue line) for scattering from a car (left) and truck (right) model. Instead of this direct simulation, a delta pulse can be used as input for the AFIT simulation to create the impulse response, which is convolved with the shortpulse signal to create the simulated microphone signal (bottom, red line). Both methods produce the same result, but the impulse response method has the benefit of creating simulated microphone data for any incident signal without re-running the entire AFIT simulation.

than an hour and only needs to be run once to find the response of the system to any arbitrary signal.

**Frequency structure of simulated microphone data**

One good use of these simulated microphone measurements is to see if the frequency content changes from the incident signals and the reflected signals. This is especially important for our experiments due to the use of coded signals to allow for the detection of reflected signals. Experimental results indicate that the structure of the backscattered signals are similar enough to the incident signals that they can be detected using correlation methods, but we can confirm this using results from the AFIT simulations.
Since our targets are moving towards us, we expect some changes in frequency content of the reflected signal due to the introduction of a Doppler shift. The Doppler shift can be calculated as

\[ \Delta f = \frac{\Delta v}{c} f_0, \]  

which relates the amount of shifted frequency \( \Delta f \) to initial frequency \( f_0 \) for speed of sound \( c \) and change in speed \( v \), which is the velocity of the receiver relative to the source (positive when source and receiver are moving toward each other). To calculate the amount of shift for our experimental measurements we assume that vehicles are driving at the speed limit of 25 mph, or roughly 11 m/s to find a frequency shift of 3.2%. This corresponds to a difference of 32 Hz for a 1 kHz incident signal and 128 Hz for a 4 kHz incident signal. We don’t expect such a small change in frequency to noticeably change the frequency of the reflected signal.

We will use the simulated microphone data to see how the frequency structure of the 100 ms 1-4 kHz linear chirp changes when the signal is reflected from the a car model. The spectrogram provides an easy assessment of the frequency structure, since a linear chirp should look like a straight line. Figure 7.28 shows the spectrogram representation of this simulated microphone data, and we see that the backscattered reflection retains the frequency structure expected for a linear chirp.

**DWFP feature extraction from simulated microphone data**

As with the experimental measurements, the goal is to classify a vehicle as one of five vehicle types based only on the backscattered acoustic signal. These time-domain pressure signals of vehicles from different classes appear very similar, even after filtering as shown in Figure 7.29 for shortpulse signals. Instead, we will perform the DWFP feature extraction from these simulated pressure data \( p(t) \). These results are
Figure 7.28: The simulated microphone signal (top) is directly simulated for a 100 ms linear chirp scattering from a car, showing the initial propagation of the incident signal and later the backscattered reflection. A time-frequency spectrogram representation (middle) clearly shows that both the initial linear chirp propagating through free space and the backscattered signal retain the frequency structure as expected. A close up view of the backscattered reflection (bottom) shows a noisier signal, but one that retains the frequency structure of a linear chirp.
not intended to be directly comparable to measured data but do provide a view into the complex feature extraction and selection process that creates a small dimensional feature vector which best describes each signal.

The most noticeable difference between the experimental and simulated data is its length and sampling rate. Our experimental measurements are acquired at 8 kHz sampling rate but AFIT simulations with the same input signal require a sampling rate of 55.4 kHz (18 μs time step). This is because the experimental sampling rate is guided by the Nyquist sampling theorem (sampling rate must be at least double the maximum frequency) but the effective sampling rate of the AFIT simulations is limited by the Courant condition (7.23). While necessary for stability, this severely restricts the upper limit on the size of the time step. The main consequence is that while we can easily capture and process several seconds of experimental data at an 8kHz sampling rate, creating simulated data several seconds long requires impractically long runtimes. Instead, our simulated \( p(t) \) signal will contain fewer than 1000 time steps, corresponding to a length of 18 ms over which the acoustic wave will propagate a total of 6.2 m in air. This is enough space for the incident signal to propagate to the vehicle model, interact with the scatterer, and reflect some portion of the sound energy back towards the measurement location.

Other differences between the simulated and measured pressure signals have a greater effect on the feature selection part of the process. Experimental datasets contain reflections from a large number of different vehicles, with multiple reflections from each vehicle in the recorded signal. Analysis of this experimental data, described in more detail in Chapter 5, requires detecting individual reflections in these recordings and accurately aligning them in time. These issues will not arise in analysis of the simulated data since our simulations look at the reflection from a single incident signal at a known distance from the scatterer.

211
Figure 7.29: The simulated pressure signal from the shortpulse signal reflecting from vehicle models and recorded at the point $z = 1$ (near the source plane) shows the difficulty in differentiating between vehicles using only the time domain data. The simulated pressure data shown here is filtered using a wavelet filter (coif3) at 3 levels to isolate the reflected signal.
As in Chapter 6, the first step in the DWFP feature extraction process is to create a fingerprint image from the simulated pressure data. Here the short length of the simulated pressure signal (\(i\ 1000\) steps) creates a fingerprint image with a few dozen fingerprints compared to thousands for the longer experimentally measured data. This will make comparisons of between classes easier to visualize. Figure 7.30 shows the labeled fingerprints images for the simulated pressure signal from the car model, highlighting the effect of filtering the raw simulated pressure signal before creating the fingerprint images. Images created from simulated pressure data from other vehicle models look similar.

![Unfiltered](image1)

![Filtered](image2)

Figure 7.30: Fingerprints are created from simulated pressure signals for the shortpulse incident signal, recorded at the point \(z = 1\) near the source plane. Here we see the how prefiltering the pressure data affects the fingerprint image - the unfiltered data (top) has more individual fingerprints than for the pressure data filtered using a wavelet filter (coif3) at 3 levels (bottom). The db3 wavelet was used in the wavelet transform. Note that the fingerprints are binary images; coloring was added to show how each fingerprint is labeled individually.

A set of one-dimensional parameter waveforms can be extracted from this binary
fingerprint image. Linearly interpolating the values of each parameter waveform across the entire length of the image preserves time-domain information and allows a direct comparison to the original pressure signal.

Comparing the parameter waveforms of measurements from different vehicles will allow us to create a set of features that best discriminates between vehicles of different classes. The automated feature selection process used for the experimental data accomplishes this by finding the points in time where the difference between parameter waveforms of different classes is maximized while the variance between parameter waveforms of the same class is minimized. However, since our simulated data contains measurements from only a single scatterer model for each vehicle class, there is by definition no interclass variance in the parameter waveforms. For these measurements, the difference between parameter waveforms of different classes is the sole measure.

Figure 7.31 shows the feature extraction and selection process for the area parameter extracted from the fingerprint image. This parameter is the area of each individual fingerprint linearly interpolated across the entire length of the image. The difference between parameter waveforms of different classes provides a metric to find the time points where the parameter values are most separated between classes. The values of each parameter waveform at these locations are then used to create an information-dense feature vector that best describes the original signal.

For our experimental measurements these feature vectors are then used in pattern classification algorithms which find patterns among the data and classifying them into appropriate categories. A comparison of known classes to these estimate classes (often in the form of a confusion matrix) provides a metric that can be used to quantify the accuracy of the entire classification process. Since our simulated data contains only one measurement point per vehicle class, this pattern classification process is
Figure 7.31: In the feature extraction process a number of one-dimensional parameter waveforms are extracted from the fingerprint image. Here we show the values of one parameter, area, for simulated microphone measurements from each of the five vehicle models. Parameters are interpolated across the entire length of the image to provide a direct comparison to the original time domain signal. This alignment in time also allows the creation of a difference signal (bottom), showing the average separation between parameter waveforms from different classes. The peaks of this signal indicate points in time where values of the parameter waveforms are most different between classes (highlighted here with a black box), and values at these points are used to create a small-dimensional feature vector.
not appropriate. Instead we will look at the performance of the feature extraction and selection process by looking at which features are chosen to comprise the final 10-feature vector.

Table 7.6 shows which individual features are selected for the feature vector when different wavelets are used to create the fingerprint image. To increase the variety of features, a total maximum of four of the same parameter were allowed. In this case, even though changing the transform wavelet in the DWFP feature extraction process creates a completely different fingerprint image, similar parameter waveforms are selected to create the 10-dimensional feature vector.

At least for the simulated data, this means that the particular choice of transform wavelet used in the DWFP process to create the fingerprint image doesn’t change which features are included in the final small-dimensional feature vector. This will potentially allow the extraction of a small number of parameter waveforms from the fingerprint image, reducing the necessary computation time and allowing feature vectors to be created on commodity hardware.
Table 7.6: The features selected for a 10-dimensional feature vectors to represent simulated pressure data (shortpulse incident signal) are fairly uniform regardless of transform wavelet used in the DWFP process to create the fingerprint image. There is a slight difference in features chosen when the data is filtered with a coif3 wavelet at 3 levels before creating the fingerprint image, with the Orientation feature being more important in the unfiltered data. Since the particular choice of transform wavelet doesn’t change which features are included in the final small-dimensional feature vector, the feature extraction process can create only those parameter waveforms, reducing computation time.

<table>
<thead>
<tr>
<th>Transform</th>
<th>Filtering</th>
<th>Chosen features</th>
</tr>
</thead>
<tbody>
<tr>
<td>haar</td>
<td>none</td>
<td>ConvexArea (x4), Orientation (x4), FilledArea (x2)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>db3</td>
<td>none</td>
<td>ConvexArea (x4), FilledArea (x3), Area (x3)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>sym5</td>
<td>none</td>
<td>ConvexArea (x4), Orientation (x4), FilledArea, Area</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>coif3</td>
<td>none</td>
<td>ConvexArea (x4), Orientation (x4), FilledArea (x2)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>meyr</td>
<td>none</td>
<td>ConvexArea (x4), Orientation (x3), FilledArea (x2), Area</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>dmey</td>
<td>none</td>
<td>Orientation (x4), ConvexArea (x3), FilledArea (x3)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x4), Area (x2)</td>
</tr>
<tr>
<td>mexh</td>
<td>none</td>
<td>Orientation (x4), ConvexArea (x3), yCentroid (x3)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>ConvexArea (x4), FilledArea (x3), Area (x3)</td>
</tr>
<tr>
<td>morl</td>
<td>none</td>
<td>ConvexArea (x4), FilledArea (x3), Area (x3)</td>
</tr>
<tr>
<td></td>
<td>coif3</td>
<td>Orientation (x4), ConvexArea (x4), FilledArea (x2)</td>
</tr>
</tbody>
</table>
Chapter 8

Future directions in sensor fusion
for mobile robotics

8.1 Sensor modalities

In this work we have investigated a number of sensor modalities that may be appropriate for mobile walking-speed robots operating in unstructured outdoor environments. A combination of short- and long-range sensors is necessary for a robot to capture usable data about its environment. Previous work focused on passive thermal infrared and air-coupled ultrasound as possible short-range sensor modalities. Our work looked at the suitability of the Microsoft Kinect as a short-range active infrared depth sensor, as well as the performance of a coffee can radar and acoustic echolocation via acoustic parametric array as long-range sensors for mobile robotics.

In particular, we have demonstrated that the most exciting feature of the Microsoft Kinect, a low-cost depth sensor, is of limited use in outdoor environments. The active illumination source in the near infrared is both limited to a range of several meters and easily saturated by sunlight so that it is mostly useful in nighttime outdoor
environments. The infrared sensor is tuned to this near infrared wavelength and provides little more information than the included RGB webcam.

The Kinect four-channel microphone array proved to be of high quality. The microphones are not spatial separated enough to allow for implementation of beamforming methods at distances over several meters and are limited to a relatively low 16 kHz sampling rate by current software, but the design of the capsule microphones and built-in noise cancellation algorithms allow for high-quality recording.

Construction of a coffee can radar showed that such devices are feasible for mobile robotics, providing long-range detection capability at low cost and in a physically small package. Since the radar signal is de-chirped to facilitate processing with a computer sound card, these measurements do not contain much useful information about scattering from the target. However, radar ranging measurements could provide an early detection system for a mobile robot, detecting objects at long range before other sensors are used to classify the object.

Another exciting possible use of the radar sensor is the creating of Synthetic Aperture Radar (SAR) images. This method to create a three-dimensional representation of the radar scattering from a target is essentially a set of ranging measurements acquired over a wide area. Normally this requires either an array of individual radar sensors or a radar that can be steered by beam forming but is a natural fit for mobile robotics since the radar sensor is in motion on a well-defined path.

The main focus of our work has been the use of acoustic echolocation as a long-range sensor for mobile robotics. Using coded signals in the audible range increases the range of the signal while still allowing for detection in noisy environments. The acoustic parametric array is able to create a tight beam of this low-frequency sound, directing the majority of the sound energy on the target. This serves the dual purpose of reducing clutter in the backscattered signal and keeping noise pollution added to
the surrounding environment to a minimum level.

As a test of this sensor modality, several thousand acoustic echolocation measurements were acquired from approaching vehicles in a variety of environmental conditions. The goal was to classify the vehicle into one of five classes (car, SUV, van, truck, or bus) based on the frontal profile. To test feasibility as a long-range sensor, vehicles were interrogated at distances up to 50 m.

Initial analysis of the measured backscattered data showed that useful information about the target under is buried deep in noise. Time-frequency representations of the data, in particular, representations created using the Dynamic Wavelet Fingerprint (DWFP) process reveal hidden information. The formal framework of statistical pattern classification allowed us to intelligently create small-dimensional, information-dense feature vectors that best describes the target. This process was able to correctly classify vehicles using only the backscattered acoustic signal with 94% accuracy.

To further improve the pattern classification process we investigated the physics of the acoustic scattering interaction. Numerical finite-difference solutions to the KZK equation allowed us to model the propagation of the nonlinear beam created by the acoustic parametric array to ensure that as much sound energy as possible is incident upon the target. Acoustic finite integration (AFIT) simulations allowed us to visualize the scattered pressure fields from any three-dimensional model at real-world scales, as well as create simulated microphone data to study the mechanisms of the complicated pattern classification routine.

8.2 Future directions in sensor fusion

Using the insight gained from experimental investigations of various sensors, we envision a new sensor package that will provide a 360° view when placed upon a
mobile robot base. A sketch of this device is shown in Figure 8.1. A custom-built octagonal parametric array (with dimensions defined via KZK numerical models) has transducers on each face so that the beam can be steered in any direction. A thermal infrared camera is placed inside the octagonal structure, pointing up towards a parabolic reflector that provides a 360° view. Atop the reflector is a circular microphone array capable of beamforming to localize sound sources. Radar sensors could be added in pairs (transmit and receive) on each side of the octagonal structure.

Figure 8.1: Concept sketches of a next generation sensor platform for mobile robots incorporate a steerable acoustic parametric and microphone array for acoustic echolocation tasks alongside a thermal infrared camera with a 360° field of view.

Future progress towards autonomous mobile robotics requires the automated interpretation of large amounts of incoming data. Using a number of complementary sensors will provide uninterrupted data streams that allow an autonomous robot to sense its environment and make informed decisions, but at the cost of processing massive amounts of information.

This problem of combining data from different sources into something useful is often
referred to as sensor fusion. The key to sensor fusion is to find a way to intelligently reduce the data to its most essential parts. Statistical pattern classification and feature selection provides a formal process to do so.
Appendix: Computer Code

We have embraced open source in our work. The open source philosophy encourages freedom and sharing, allowing projects to become greater than any one individual could create on their own. Whenever possible, we have chosen to use open-source software. The computer controlling rMary data acquisition runs GNU/Linux, OpenScad and MeshLab are used to create and manipulate three-dimensional solid object models, Audacity allows us to easily label events in recorded audio signals, and VisIt creates three-dimensional visualizations of simulation results.

The open source philosophy extends past free software. Three-dimensional object models are freely shared on the website Thingiverse, and full plans and instructions freely available online allowed us to create a low-cost coffee can radar system. In some cases, this has even provided us with access to hardware features and data streams locked down by device manufacturers, as was the case with the OpenKinect drivers for the Microsoft Kinect. Continuing to contribute to the open source community should be an important component of any future work.

The code provided here was tuned to work on specific hardware. Current copies of working code may be obtained by contacting the author.

rMary data capture

#!/bin/bash
# rMary_getdata
# Overarching rMary data collection script, for a single vehicle:
# Playback of pulse via audio-out (choose short or long)
# Recording kinect audio via usb (4-channel mic array as separate wav files)
# Recording radar (ranging vs time) via audio-in
#
# Usage: rMary_getdata <output signal> <Recording length (sec)>
# Eric A. Dieckman (WM)
# 30 Jan 2013
# Last edited: 31 Jan 2013 EAD

now=\$(date +%Y%m%d-%H%M%S)

mkdir ~/Desktop/rMary_Data/$now
cd ~/Desktop/rMary_Data/$now

# Decide which signal to play on the param array
if [ "$1" = 'long' ]
then

223
PLAYFILE="/Desktop/rMary_Data/pulsechirp_1k-4k_250_750" # 9 sec
elif [ $1 = 'short' ]
then
PLAYFILE="/Desktop/rMary_Data/pulsechirp_1k-4k_100_900" # 9 sec
elif [ $1 = 'nosound' ]
then
RECAMS=1
else
echo "Call with argument long, short, or nosound"
fi

# Grab Kinect image, play signal, record audio and radar
"/kinect_devel/build/bin/grab_kinect_data" # grabs Kinect RGB and depth image
"/kinect_devel/build/bin/grab_kinect_data --grabir" # grabs Kinect raw IR and depth image
if [ $RECAMS ]
then
  echo "Recording ambient signal only"
"/libfreenect/build/bin/wavrecord > kinect.txt & pid=$! & # record sound with Kinect
  rec radar.wav trim 0 $2 & # Save radar data
else
  play $PLAYFILE.wav trim 0 $2 & # play sound
  rec -c 2 radar.wav trim 0 $2 & # Save radar data
  "/libfreenect/build/bin/wavrecord > kinect.txt & pid=$! & # record sound with Kinect
fi

# mv ~/Desktop/kinect_out/kinect_ir.png kinect_ir.png
mv "~/Desktop/kinect_out/kinect_rgb.png kinect_rgb.png
mv "~/Desktop/kinect_out/kinect_normdepth.png kinect_depth.png

# Kills processes after max amount of time - stop earlier using ESC and CTRL+C
sleep $2
kill -9 $pid

echo "now
ls -lh

cd "~/Desktop/rMary_Data

rMary data compilation

% rMary_alldata_compile.m
% Compile data from specified classes, create data structures for 'rMary_alldata'
% All necessary files are in /rMary/alldata and subdirs
% %
% Usage:
% % In:
% % Out:
% % Dependencies: wavchuncksfix

% Eric A. Dieckman (WAM)
% 11 February 2013
% Last edited:29 April 2013 EAD

function rMary_alldata_compile
% Data compilation settings:
% basedir = '~/Research/8-rMary/alldata/'; % single directory of timestamped folders
("oldkinect" is subdirectory)

%manifest = '~/home/eric/Dropbox/Work/Matlab/WN/rMary_manifest_5-750_HD.txt'; % manifest of timestamped folders and associated classes
%manifest = '~/home/eric/Dropbox/Work/Matlab/WN/rMary_manifest_10-750_HD.txt';
%manifest = '~/home/eric/Dropbox/Work/Matlab/WN/rMary_manifest_100-900_HD.txt';
%manifest = '~/home/eric/Dropbox/Work/Matlab/WN/rMary_manifest_250-500_HD.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-500_NHO.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-500_all.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-1000_NHO.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-1000_all.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-1000_comb_NHO.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-1000_comb_all.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-comb_NHO.txt';
%manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-comb_all.txt';
manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-comb_NHO.txt';
manifest = 'home/eric/Dropbox/Work/Matlab/WM/rMary.manifest.250-comb_all.txt';

pulse = '250-comb'; % pulsetype (to save to structure) options: '5-750', '10-750', '100-900', '250-600', '250-1000', '250-comb'

classesstodo = ['c' 's' 'v' 't' 'b' 'm' 'o']; % tags to include (c s v t b m o), include combos here
taglimit = 7; %'equaltags'; % options: 'alltags', 'equaltags', number
channel = 3; % pick a channel of kinect data to use

doresamp = 0; % whether or not to resample to min

saveradar = 0; % whether or not to save radar data to structure

saveimage = 0; % whether or not to save kinect image data to structure - NOT IMPLEMENTED

savename = '20130429-fiveclass_limited'; % will go in 'rMary_compdata_pulsetype_SAVENAME'.mat

origfs = 16000; % original sampling rate

if strcmp(pulse,'100-900') % checks if old or new data
    olddata = 0;
else
    olddata = 1;
saveradar = 0;
end

% Create subset of folders to pull data from
[timestamp knownclass] = textread(manifest, '%XsXs'); % all possible data

for pullidx = 1:length(classesstodo)
    currentclass = char(classesstodo(pullidx));
    alldata(pullidx).knownclass = currentclass;
    if length(currentclass)>1 % combined class
        topull = 0;
        for combineidx = 1:length(currentclass)
            topull = topull + strcmp(currentclass(combineidx).knownclass);
        end
        topull = logical(topull);
    else
        topull = strcmp(currentclass,knownclass); % which data to pull for each class (logical)
    end
    alldata(pullidx).timestamp = timestamp(topull); % folder names containing data for each class
    alldata(pullidx).numdata = sum(topull); % how much data for each class
    clear topull currentclass;
end

% Pick subset of data from larger classes (if option is selected in settings)
if strcmp(taglimit,'alltags')
    disp('Keeping all data from all classes (probably unequal amounts!)');
    selectdata = alldata;
elseif strcmp(taglimit,'equaltags') % equal number of tags (based on least populous group)
    disp('Keeping equal amount of data for all classes, based on lowest amount');
    [minnum minidx] = min([alldata.numdata]); % find which class has the least data
    for idx = 1:length(classesstodo)
        pick = randperm(alldata(idx).numdata); % randomized list
    end
selectdata(idx).timestamp = alldata(idx).timestamp(pick(1:minnum)); % randomly
pull subset of data
selectdata(idx).numdata = minnum;
selectdata(idx).knownclass = char(classestodo(idx));
selectdata(idx).pulsetype = pulse;
end
else % equal number of tags, based on given number
  disp('Keeping equal amount of data for all classes, based on user input');
  if sum(~(taglimit < [alldata.numdata]))
    disp(['Not that much data in a class! Make taglimit less than ' num2str(min([alldata.numdata]))]);
    return;
  else
    for idz = 1:length(classestodo)
      pick = randperm(alldata(idx).numdata); % randomized list
      selectdata(idx).timestamp = alldata(idx).timestamp(pick(1:taglimit));
      % randomly pull subset of data
      selectdata(idx).numdata = taglimit;
      selectdata(idx).knownclass = char(classestodo(idx));
      selectdata(idx).pulsetype = pulse;
    end
  end
end

% Pre-calculate data size - Can add option to halt here if becomes necessary in the future
totnumdata = sum([selectdata.numdata]);
disp(["Structure will contain data from ' num2str(totnumdata) ' measurements"]);
if saveradar == 1
disp(['Estimated structure size: ' num2str(0.9*totnumdata) ' mb']);
else
  disp(['Estimated structure size: ' num2str(0.34*totnumdata) ' mb']);
end

% Now fix kinect wav file, save to structure
for idx = 1:length(selectdata)
  for dataidx = 1:selectdata(idx).numdata
    if olddata == 1 % reading in from structures, not raw wav files
      tempdata = load(['oldkinect/' char(selectdata(idx).timestamp(dataidx)) '.mat']); % load structure
      if doresamp == 1 % resample
        if strcmp(pulse,'short') || strcmp(pulse,'long') % can resample to 8 kHz
          selectdata(idx).kinect(dataidx).data = resample(tempdata.splitdata(channel).data,1,2); % save kinect data to structure
          selectdata(idx).fs = resample(origfs,1,2);
        else % resample to 10 kHz
          selectdata(idx).kinect(dataidx).data = resample(tempdata.splitdata(channel).data,5,8); % save kinect data to structure
          selectdata(idx).fs = resample(origfs,5,8);
        end
      else % don't resample
        selectdata(idx).kinect(dataidx).data = tempdata.splitdata(channel).data;
        % save kinect data to structure
        selectdata(idx).fs = origfs;
      end
    else % new data, read in from raw wav files
      tempkinectname = [char(selectdata(idx).timestamp(dataidx)) '/channel' num2str(channel) '.wav'];
      if exist(tempkinectname) == 2; % check to ensure kinect data present
wavchunksizerfix(tempkinectname); % fix Kinect wav file
tempload = wavread(tempkinectname);

if doresamp == 1 % resample
    if strcmp(pulse,'short') || strcmp(pulse,'long') % can resample to 8 kHz
        selectdata(idx).kinect(dataidx).data = resample(tempload,1,2);
        % save Kinect data to structure
        selectdata(idx).fs = resample(origfs,1,2);
    else % resample to 10 kHz
        selectdata(idx).kinect(dataidx).data = resample(tempload,5,8);
        % save Kinect data to structure
        selectdata(idx).fs = resample(origfs,5,8);
    end
else % don't resample
    selectdata(idx).kinect(dataidx).data = tempload; % save Kinect data to structure
    selectdata(idx).fs = origfs;
end

selectdata(idx).kinect(dataidx).channel = channel; % save channel to structure
clear tempload;
else
    disp(['No Kinect data present in ' char(selectdata(idx).timestamp(dataidx))]);
end

if saverradar == 1
    tempradarname = [char(selectdata(idx).timestamp(dataidx)) '.radar.wav'];
    if exist(tempradarname) == 2 % ensure radar data is present
        selectdata(idx).radar(dataidx).data = wavread(tempradarname); % save Kinect data to structure
    else disp(['No radar data present in ' char(selectdata(idx).timestamp(dataidx))]);
    end
end

if saveimages == 1
    disp('Feature not yet implemented - allows saving info from Kinect images to structure');
end
end

disp(['Data loaded from class ' num2str(idx) ' of ' num2str(length(selectdata))]);

% Save structure
save('compileddata/rMary_compdata.' pulse '_' savename '.mat','selectdata');
cd(nowdir)

rMary pattern classification

% rMary_alldata.m
% rMary analysis script for param array, Kinect audio, and radar data - parallelized
% Based on 'rMary_parkingLot.m'
% Reads in compiled data structure (from 'rMary_alldata_compile'), dwpf/feature creation, feature selection
%
% Usage:
% In:
% Out:
Run first:

function rMary_alldata
X USER INPUT:
savedir = '~/Research/8-rMary/alldata/classdata/'; % directory to save output files

% Data pre-analysis
parlabs = 4; % max number of parallel processors to use
howtosplit = 'xcorr'; % how to split acoustic data - 'envelope', 'xcorr'
usekinectavg = 'random'; % which cut data should be used in fingerprinting - 'average', 'random', 'all'
datatonal = 'acoustic'; % which data to use in analysis - 'acoustic', 'radar', 'both'

% Fingerprint filtering settings:
filmethod = 'filtandwindow'; % options: 'filt', 'filtandwindow', 'window', 'none'
wvtpf='coif3'; % filtering wavelet Db3
numlevels=6; % number of levels for filtering
swtdetoremove=1.6; % Number of details to remove

% Wavelet transform settings:
wvt='db3'; % Wavelet transform Db3
ne= 50; % Number of scales
normconstant=1; % Normalization constant
numsllices= 15; % Number of slices
slicethickness=0.03; % Slice thickness

% Feature extraction settings
saveimages = 1; % boolean, whether or not to save fp image to structure
fullorred = 'reduced'; % Output full or reduced fingerprint stats - options 'full', 'reduced'
solidity_range = [0.3 0.6]; % Allowable range for solidity (for 'reduced' output)
savefpstruct >1; % Save fingerprint structure

% Feature selection settings
peakdetect • 'joint'; % method to pick top points - 'separate' or 'joint'
viewselected = 1; % view selected points on mean/standard deviation or jointdiff plot
selectnfeats = 5; % keep this many top points in feature selection
topnfeats = 25; % keep this many top points overall

% =========== Wavelet options ===========
% Haar
% Daubechies
% Symlets
% Coiflets
% Meyer
% DMeyer
% Mexican_hat
% Morlet
% ================

XX MAIN PROGRAM

tic
time1 = toc; % for timing

[fname, fpath] = uigetfile('~/Research/8-rMary/alldata/compileddata', 'Select compiled data file'); % select file
X[fname, fpath] = uigetfile('~/Dropbox/Work/W*M/Research/rMary', 'Select compiled data file'); % laptop
data = load([fpath fname]); % load data
data = data.selectdata;
pulsetype = data(1).pulsetype;
fs = data(1).fs;
if fs < 16000
    disp('Please use non-resampled (16kHz) data at this time');
    return;
end

nbrclasses = length(data);
if nbrclasses < paralabs % no need to use more parallel
    paralabs = nbrclasses;
end

% Filter, split, and fingerprint acoustic data —-
if strcmp(datatoanal,'acoustic') || strcmp(datatoanal,'both')
    X Bandpass and split Kinect audio data into new structure — removes data that can't
    be split
    matlabpool(paralabs); % parallel
    parfor classidx = 1:nbrclasses
        count = 1;
        notcutidx = 0;
        for idx = 1:length(data(classidx).kinect) % number of measurements in each class
            tempcut = bpxcorr(data(classidx).kinect(idx).data, pulsetype, fs, howtosplit); % returns cut data structure (with nocuts filtered out)
            if tempcut == 9999 % couldn't cut data
                notcutidx = [notcutidx idx]; % keep track of which data can't be split
            else
                kinectdata(classidx).measurement(count).cutdata = tempcut;
                count = count + 1;
            end
        end
        notcut(classidx).idx = notcutidx;
        disp(['Class ' num2str(classidx) ' of ' num2str(nbrclasses) ' split']);
        disp(['Reduced to ' num2str(count-1) ' of ' num2str(length(data(classidx).kinect)) ' measurements']);
        disp(['No cut data for measurements: ' num2str(notcutidx)]);
    end
end

% viewcut(kinectdata, 1); % uncomment to view cut data

% Make fingerprints and extract features from acoustic data
disp('Starting DWFP and feature extraction (this will take a while)');
parfor classidx = 1:nbrclasses
    for idx = 1:length(kinectdata(classidx).measurement)
        [nbrcuts cutl] = size(kinectdata(classidx).measurement(idx).cutdata); % number of cuts for each measurement
        switch usekinectavg
            case{'average'} % average all cut data (single fp per measurement)
                if nbrcuts == 1 % don't need to average
                    compare(classidx).fpdata(idx) = fingerprint(kinectdata(classidx).
                        measurement(idx).cutdata, fs, saveimages, filtmethod,...
                            wvtpf, numlvls, wdtoremove, wvt, ns, normconst, numslcs,
                            slicethickness, fullorred, solidity.range); % create fingerprints, extract feats
                else % average
                    compare(classidx).fpdata(idx) = fingerprint(mean(kinectdata(
                        classidx).measurement(idx).cutdata), fs, saveimages, filtmethod,
                            wvtpf, numlvls, wdtoremove, wvt, ns, normconst, numslcs,
                            slicethickness, fullorred, solidity.range); % create fingerprints, extract feats
                end
            case{'random'} % choose one cut data (single fp per measurement)
end

229
compare(classidx).fpdata(idx) = fingerprint(kinectdata(classidx).
measurement(idx).cutdata(randi(nbrcuts,1),:), fs, savemeasure,
filmethod, wtpf, numlvls, svdremove, wvt, ns, normconstant,
umslice, slicethickness, fullreord, solidity_range); % create
fingerprints, extract feats

case('all') % use each cut data (multiple fp per measurement)
disp('Using multiple cut data from a single measurement is not
currently supported');
end

matlabpool close; % NEED MORE LATER?

if savefpstruct == 1
    save([savedir 'fpdata'], 'compare');
end

disp(['Acoustic feature extraction completed -- Elapsed time: ', num2str(
    (toc-time)/60), ' min']);

%% Analyze radar data — %
if strcmp(datatoanal,' radar') || strcmp(datatoanal,'both')
    % DO RADAR ANALYSIS
    disp('Radar analysis not yet implemented');
end

%% Feature selection — %
disp(['Now beginning feature selection for ' num2str(nbrclasses) ' classes']);

% Cell array of known classes
knownclasses = {);
for classidx = 1:nbrclasses
    for idx = 1:length(kinectdata(classidx).measurement)
        knownclasses{end + 1} = data(classidx).knownclass;
    end
end

[topfeas selected_feats ] = featcompare(compare, knownclasses, selectnfeats, topnfeats,
pkddetect, viewselected);
disp(['Top ' num2str(selectnfeats) ' time points with best separation compiled for ' num2str(length(selected_feats.timepts)) ' features' 'Elapsed time: ', num2str(
    (toc-time)/60), ' min']);
disp('Features selected are from ');
char(selected_feats.feats(selected_feats.topfeas.indices(:,1))

%% Save data to do pattern classification in rMary.classify — % Separate data into
training/testing sets; do pattern classification!
save([savedir 'classdata.mat'], 'topfeas', 'selected_feats');

% Pre-analysis of acoustic data - bandpass filter, xcorr with pulse

function splitdata = bpxcorr(data, pulsetype, fs, howtosplit)
% Input- ' data' is n x 1 array (from a single measurement)
fkhznyq = fs/1000/2; % half sampling rate in kHz (Nyquist)
switch pulsetype % possible: '100-900', '250-500', '250-1000', '250-comb', '10-750',
    '5-750'
    case('100-900')
        pulse = chirp(0:1/fs:0.100,1000,0.100,4000,'linear'); % 100ms pulse
        Wp = [1 4]/fkhznyq; Ws = [0.5 4.5]/fkhznyq; % bandpass from 1-4 kHz, 500 Hz

    end
end
stopbands
splitsec = 1; % length to split data (sec)
case('250-500', '250-1000', '250-coeb')
pulse = chirp(0:1/fs:0.250,100,0.250,5000,'linear'); % 250ms pulse, diff freq range
Wp = [0.1 4]/fskhznyq; Ws = [0.06 5.5]/fskhznyq; % bandpass from 100 Hz-5 kHz;
50, 500 Hz stopbands % DO THIS, or just all at 1k-4k?
splitsec = 1;
case('10-750')
pulse = chirp(0:1/fs:0.010,1000,0.010,4000,'linear'); % 10ms pulse
Wp = [1 4]/fskhznyq; Ws = [0.6 4.5]/fskhznyq; % bandpass from 1-4 kHz, 500 Hz stopbands
splitsec = 0.85;
case('5-750')
pulse = chirp(0:1/fs:0.005,1000,0.005,4000,'linear'); % 5ms pulse
Wp = [1 4]/fskhznyq; Ws = [0.5 4.5]/fskhznyq; % bandpass from 1-4 kHz, 500 Hz stopbands
splitsec = 0.85;
end

Filter params
Rp = 3; Rs = 40; % max 3dB ripple in passband, 40 dB atten. in stopband
[n,Wn] = buttord(Wp,Ws,Rp,Rs);
[b,a] = butter(n,Wn);

filtdata = filter(b,a,data);
XFiltdata = filtdiag(tukeywin(length(data), 0.26)); % Tukey window

switch howtosplit
  case('Envelope') % select start points from peaks of envelope signal
    rectfilt = abs(hilbert(filtdata)); % get envelope
    envcutoff = 20; % lowpass filter cutoff for envelope (Hz)
    [b2,a2] = butter(6,envcutoff/fs/2,'low'); % lowpass filter envelope
    smooth = filtfilt(b2,a2,rectfilt);
    smooth = smooth - mean(smooth); % detrend
    starts = find(smooth>0.65*max(smooth)); % Find start pts (65% of max value)
  case('xcorr')
    corrdata = xcorr(pulse,filtdata); % xcorr
    corrdata = corrdata(1:length(filtdata) + length(pulse) -1);
    starts = find(corrdata>0.7*max(corrdata)); % Find start pts (70% of max value)
end

% starts = starts - length(pulses); % corrects for xcorr offset
splitstarts = split_contiguous(starts, 100); % split into contiguous chunks
starthere = zeros(1,length(splitstarts)); % initialize
for idx = 1:length(splitstarts)
pulsestartpt(idx) = ceil(mean(cell2mat(splitstarts(idx)))); % avg each value
end

% some measure of xcorr peaks
xfimfig = plot(filtdata(1,:)/max(filtdata(1,:))); hold on; plot(corrdata/max(corrdata), 'r'); plot(starthere,'g'); hold off; axis tight; pause;
if isempty(starthere)
  disp('No start points found!');
splitdata = 9999; % set to error code
else
    for idx = 1:length(startthere)
        if startthere(idx) + cutl < length(data) % ensures no eof problems
            splitdata(idx,:) = filtdata(startthere(idx):startthere(idx) + cutl);
            % pulsestartpt(idx) = pulsestartpt(idx); % actual start point of pulse (NOT
            % WHERE splitdata.data STARTS)
        end
    end
end

%%%%%%%%% View cut data %%%%%%%%
function viewcut(cutdatastruct, classnbr)
    % View all cut data from each measurement for a particular class, alongside the average
    for idx = 1:length(cutdatastruct(classnbr).measurement)
        disp(['Measurement ' num2str(idx) ' of ' num2str(length(cutdatastruct(classnbr).
            measurement))]);
        [multicut cutl] = size(cutdatastruct(classnbr).measurement(idx).cutdata);
        for multidx = 1:multicut
            subplot(multicut+1,1,multidx); plot(cutdatastruct(classnbr).measurement(idx).
                cutdata(multidx,:))
        end
        subplot(multicut+1,1,multicut+1); plot(mean(cutdatastruct(classnbr).measurement(idx).
            cutdata)); title('Average');
        pause;
    end
end

%%%%%%%%% Fingerprint subfunction %%%%%%%%
function [fpout] = fingerprint(rawdata, fs, saveimages, filtmethod, wvtpf, numlvls,
    swdttoremove, wvt, na, normconstant, numalices, slicethickness, fullorred, solidity_range)
    % Performs DWFP and extracts features from fingerprints
    % In:
    % Out: 'fpout'- fingerprint structure
    name = [] ; % can add identifying name to structure if desired
    leftwin = 1;
    rightwin = length(rawdata);
    % Filter and transform
    rawdata = rawdata((1:cutl)+rem(length(rawdata),2*numlvls)); % clip raw data to
    % appropriate size for wavelet transform
    if rightwin > length(rawdata); % make sure if data is clipped, we don't try to access
        rightwin = length(rawdata); % outside of that range
    end
    switch filtmethod
    case('filt')
        [sva,svd] = svd(rawdata, numlvls, wvtpf); % Single wavelet transform
        sva(svdttoremove,:) = 0; % Filter out details
        filtdata = isvt(sva, svd, wvtpf); % Inverse wavelet transform
        filtdata = filtdata(:,);
        case('filtandwindow') % Filter and window
            [sva,svd] = svd(rawdata, numlvls, wvtpf); % Single wavelet transform
            sva(svdttoremove,:) = 0; % Filter out details
            filtdata = isvt(sva, svd, wvtpf); % Inverse wavelet transform
            filtdata = filtdata(:,);
            filtdata = filtdata.*tukeywin(length(filtdata), 0.25); % Tukey window
        case('window') % window only
            filtdata = rawdata;
            filtdata = filtdata(:,);
            filtdata = filtdata.*tukeywin(length(filtdata), 0.25); % Tukey window
        case('none') % rawdata only
            filtdata = rawdata;
        end
    end

232
% Add raw and filtered data to output structure, also name, fs, and dvfp settings
fpout.rawdata = rawdata;
fpout.filtdata = filtdata;
fpout.fs = fs;
fpout.name = name;
fpout.dvfpinfo = {'Filtering info: method-' filtmethod ', wavelet-' wvtpf ', levels-' num2str(numlevels) ', details to remove-' num2str(num2remove) 10 ...'
  'Transform info: wavelet-' wv ', scales-' num2str(ns) ', slice thickness-' num2str(slicethickness) 10 ...'
  'General info: normalization-' num2str(normconstant) 10 ...'
  'Feature extraction info: stats type-' fullorred ', solidity range-' num2str(solidity_range(1)) '-'  num2str(solidity_range(2)));

% Add some stats from the raw and filtered data
fpout.wavstats = getwavstats(fpout.rawdata, fpout.filtdata, fs);

% Perform DVFP separately on 'peaks' (positive coefficients) and 'valleys' (negative coefficients) - can combine later into single array (add grey multiplier)
fppeaks = doonethumbprint(filtdata(leftwin:rightwin), wvt, ns, normconstant, numalices, slicethickness, 2); % peaks (white)
fppeaks = flipud(fppeaks); % put origin in lower left hand corner

fpvalleys = doonethumbprint(filtdata(leftwin:rightwin), wvt, ns, normconstant, numalices, slicethickness, 3); % valleys (grey)
fpvalleys = flipud(fpvalleys);

% Get image stats from fingerprint -> Calls subfunction 'getfpstats'
[fppeaks.stats, fppeaks.ridgecount, fppeaks.stats.image] = getfpstats(fppeaks, solidity_range, fullorred);
[fpvalleys.stats, fpvalleys.ridgecount, fpvalleys.stats.image] = getfpstats(fpvalleys, solidity_range, fullorred);

% Add images to structure (or not) -- always add stats and ridgecount to structure
fpout.fppeaks.stats = fppeaks.stats;
fpout.fpvalleys.stats = fpvalleys.stats;
fpout.fppeaks.ridgecount = fppeaks.ridgecount;
fpout.fpvalleys.ridgecount = fpvalleys.ridgecount;

switch saveimages
  case{0>
    disp('Not saving fpimages to structure');
  case{1>
    disp('Saving fpimages to structure');
    fpout.fppeaks = fppeaks;
    fpout.fpvalleys = fpvalleys;
    fpout.fppeaks.stats_image = fppeaks.stats_image;
    fpout.fpvalleys.stats_image = fpvalleys.stats_image;
end

disp('Fingerprint created');

% XXXXXX Fingerprint subsubfunctions XXXXXX
function [fpstats, fpridgecount, fpstatsimage] = getfpstats(fpimage, solidity_range, fullorred)
  % In: 'fpimage'- fingerprint image, 'solidity_range'- to reduce number of prints, 'fullorreduced'- output 'full' or 'reduced'
  % Out: 'fpstats'- linearly interpolated structure of stats from fingerprint images (relabeled so cocentric objects are the same), either full or reduced, 'fpstatsimage'- fingerprint
  % image showing labeling
  % Dependencies: 'lin_interp', 'ridge_count', 'labelnearlyconnected', 'getxcentroids' subfunctions
  siglength = length(fpimage);
  fpbin = im2bw(fpimage); % converts to binary

233
if length(unique(fpimage)) > 2 % contains both peaks and valleys -> split into 2 images
to find centroids better -> won't apply here!
fpbinpeak = im2bw(fpimage,0.7); % only peak values (is in fpimage)
fpbinval = im2bw(fpbin - fpbinpeak); % get valleys by subtraction

[Lpeak, numpeak] = labelnearlyconnected(fpbinpeak); % label individual images
[Lval, numval] = labelnearlyconnected(fpbinval);

peakcents = getcentroids(Lpeak); % get centroids of newly-relabeled images
valcents = getcentroids(Lval);

if peakcents(1) < valcents(1) % so first print is labeled correctly
    allcents = [peakcents valcents]; % get ready to put the images together again
tomod = numpeak; % number of fingerprints in first image
firstlook = Lpeak;
firstwrite = Lpeak;
secondlook = Lval;
secondwrite = Lval;
elseif valcents(1) < peakcents(1)
    allcents = [valcents peakcents];
tomod = numval;
firstlook = Lval;
firstwrite = Lval;
secondlook = Lpeak;
secondwrite = Lpeak;
end

[sortedcents, sortedcentsidx] = sort(allcents); % fingerprint centroids in right order
split = sortedcentsidx - tomod; % negative values are from first set, positive values give position in second set
for idx = 1:length(sortedcents)
    if split(idx) > 0
        secondwrite(find(secondlook==split(idx))) = idx;
    else
        firstwrite(find(firstlook==sortedcentsidx(idx))) = idx;
    end
end

newL = firstwrite + secondwrite; % re-combine the label images
else
    [newL, num] = labelnearlyconnected(fpbin); % relabel fingerprint
end

%newL = double(newL);
fullstats = regionprops(newL, 'all'); % compute new stats
% Choose which to output
switch fullorred
    case{'full'} % output full stats - no reason to calculate more
        disp([num2str(length(fullstats)) ' individual fingerprints']);
        fpstats = lin_interp(fullstats,siglength); % output linearly-interpolated stats
        fpridgecount = ridge.count(newL);
        fpstatsimage = newL;
    case{'reduced'} % Remove fingerprints that aren't useful (too small)
        redL = newL; % copy L
tokestead = [fullstats.Solidity] > solidity_range(1) & [fullstats.Solidity] <
            solidity_range(2); % keep the is
torem = ~tokestead; % keep the is
        temp = 1:length(torem);
        remidz = temp(torem); % indices of features to remove
        for idx = 1:length(remidz)
```matlab
[a b] = find(redL == remidx(idx));
redL(a:b) = 0; % remove fingerprints
end
redstats = regionprops(redL, 'all');
count = 1;
for idx = 1:length(redstats)
    if ~isnan(redstats(idx).Centroid) % remove empty entries from redstats
        newredstats(count) = redstats(idx);
        count = count + 1;
    end
end
disp(['num2str(length(newredstats)) ' individual fingerprints, reduced from ' num2str(length(fullstats))]);
fpstats = lin_interp(newredstats,siglength);
fpridgecount = ridge_count(redL);
fpstatsimage = redL;
end
function [L, num] = labelnearlyconnected(bw)
bwl = labelmatrix(bwconncomp(bw)); % original label
figure; imshow(label2rgb(bwl, 'jet', [.7 .7 .7], 'shuffle'));
bw2 = bwdist(bwl) <= 2.5; % pixels less than 5 apart will be considered the same
cc = bwconncomp(bw2); % find new connected components
num = cc.NumObjects;
L = labelmatrix(cc); % relabel
figure; imshow(label2rgb(L, 'jet', [.7 .7 .7], 'shuffle'));
L('bw') = 0; % remove the added points
end
function [cents] = getxcentroids(labeledimage)
tempcents = regionprops(labeledimage, 'Centroid'); % get centroids of newly-relabeled images
cents = [tempcents.Centroid]; % array of centroids
cents = abs(cents(1:2:end)); % but we only need the odds (x-centroids)
end
function ridge_count = ridge_count(fpimage)
%Counts ridges in fingerprint image - assumes no ridges extend to very top of image
% In: 'fpimage' image
% Out: 'ridge_count' vector - plot with stairs(ridge_count)
[row,col] = size(fpimage);
ridge_count = zeros(1,col);
for j = 1:col;
    for i = 1:row;
        if fpimage(i-1,j) == 0
            ridge_count(j) = ridge_count(j) + 1;
        end
    end
end
ridge_count = row - 1 - ridge_count; % flip so that no fingerprints = 0 ridge count
end
function [lin_fpstats] = lin_interp(fpstats, siglength)
% linearly interpolate stats to get continuous signal
% In: 'fpstats' array, 'siglength' - length of raw signal (will interpolate to this length)
% Out: 'lin_fpstats' - structure of linearly interpolated states -> only y-values b/c
% x-values are just sample number (1:1:length(data))
% Currently saved stats: Area, FilledArea, Extent, ConvexArea, EquivDiameter,
% Solidity, yCentroid, major/minor axis lengths, eccentricity, orientation, Euler Number, orig_xpos
disp('Interpolating');
xpos = [fpstats.Centroid]; % get centroids
xpos = floor(xpos(1:2:end)); % want only odds (x-centroids), make sure integer
xinterp = 1:1:siglength; % interpolate the signal to length of original samples
```

235
% Catch non-distinct x error
if length(unique(xpos)) ~= length(xpos)
    disp('Fixing non-distinct x-values');
    [xpos, unqi] = unique(xpos);
    fpstats = fpstats(unqi); % remove any problem points
end

% Area stats
lin.fpstats.Area = interp1(xpos, [fpstats.Area], xinterp); % area
lin.fpstats.FilledArea = interp1(xpos, [fpstats.FilledArea], xinterp); % filled area
lin.fpstats.Extent = interp1(xpos, [fpstats.Extent], xinterp); % extent
lin.fpstats.ConvexArea = interp1(xpos, [fpstats.ConvexArea], xinterp); % convex area (area of smallest convex polygon that contains the area)
lin.fpstats.EquivDiameter = interp1(xpos, [fpstats.EquivDiameter], xinterp); % Equivalent diameter
lin.fpstats.Solidity = interp1(xpos, [fpstats.Solidity], xinterp); % Solidity

% Other stats
temp.ycent = [fpstats.Centroid];
lin.fpstats.yCentroid = interp1(xpos, temp.ycent(2:2:end), xinterp); % y-centroid
lin.fpstats.MajorAxisLength = interp1(xpos, [fpstats.MajorAxisLength], xinterp); % major axis length
lin.fpstats.MinorAxisLength = interp1(xpos, [fpstats.MinorAxisLength], xinterp); % minor axis length
lin.fpstats.Eccentricity = interp1(xpos, [fpstats.Eccentricity], xinterp); % eccentricity
lin.fpstats.Orientation = interp1(xpos, [fpstats.Orientation], xinterp); % orientation
lin.fpstats.EulerNumber = interp1(xpos, [fpstats.EulerNumber], xinterp); % Euler number
lin.fpstats.orig.xpos = xpos; % vector of original x-centroids

function [wavstats] = getwavstats(ravdata, filtdata, fs)
    % In: 'ravdata', 'filtdata'
    % Out: 'wavstats'- structure of stats from wav files (raw and filtered)

    % Means
    wavstats.rawmean = mean(ravdata);
    wavstats.filtmean = mean(filtdata);

    % Power spectral density
    h = spectrum.welch;
    wavstats.rawpsd = psd(h,ravdata,'Fs',fs);
    wavstats.filtpsd = psd(h,filtdata,'Fs',fs);

function [thumbprint] = doonethumbprint(datain, wvt, ns, normconstant, numslices, slicethickness, valleyorpeaks)
    % In: 'datain'- raw data, 'wvt'- wavelet name, 'ns'- number of scales to use during wavelet decomposition, 'normconstant'- normalization constant, % 'numslices'- number of slices, 'slicethickness'- thickness of slice (is .1 ), 'valleyorpeaks'- 1 for both, 2 for just peaks, 3 for just valleys
    % Out: 'thumbprint'- output image

    cfX = cvt(datain, 1:ns, wvt); % get continuous wavelet transform coefficients
    maxes = max(max(abs(cfX)))*normconstant; % find max (of all coefficients) to normalize scale
    cfX = cfX./maxes; % normalize coefficients
    temp(1:ns,1:length(datain))=0; % initialize image

switch valleyorpeaks
    case 1, slicelocations = [-1:(1/numslices):-((1/numslices)-(1/numslices)) (1/numslices):((1/numslices)+(1/numslices))]; % both valley and peaks
    case 2, slicelocations = (1/numslices):((1/numslices)); % peaks
    case 3, slicelocations = -1:(1/numslices); % valleys
end

for si = 1:length(slicelocations) % slice!
if eliciations(sl)='0 X Set ridges to white
    xpeak = find((cfX=(eliciations(sl)-(slicethickness/2)) & (cfX=(eliciations(sl)+(slicethickness/2)) ));
    tem(xpeak) = 1;
end
end
thumbprint = tem; % output thumbprint

Feature selection subfunction XXXX
function [ topfeats selected_feats ] = featcompare(compare, tagnames, featdepth, topfeats, peakdetect, viewselected)
% Does feature selection by averaging, then find where mean_diff is max and stdev is min between different tags
% In:
% Out: selected_feats.timepts is size(nbrfeats x featdepth) that contains time points
% where avgdata from different tags are most separated

% features to average (all of same length)

nbrfeats = length(feats);
 nbrtags = length(compare);
datai = length(compare(1).fpdata(1).filtdata);
% get length of data
avgdata = zeros(nbrfeats, nbrtags, datai);
stddev = zeros(nbrfeats, nbrtags, datai);

% Initialize avg data array - size is (nbr feats x nbr tags x datalength)
for featidx = 1:nbrfeats X go through all features
    for idx = 1:nbrtags
        toavg = compare(idx).fpdata;
        [a,b] = size(toavg);
        temp = zeros(b,datai); % initialize
        for avgidx = 1:b
            temp(avgidx,:) = eval(['toavg(avgidx).' featname]); % build up temporary array to average
        end
        temp(isnan(temp)) = 0; % remove any nans before we average!
        avgdata(featidx,idx,:) = mean(temp,1);
        stddev(featidx,idx,:) = std(temp,0,1);
    end
end
% Normalize
avgdata(featidx,idx,:) = avgdata(featidx,idx,:)/max(abs(avgdata(featidx,idx,:)));
stddev(featidx,idx,:) = stddev(featidx,idx,:)/max(abs(stddev(featidx,idx,:)));
allcomp(idx,featidx).data = temp; % save all feature data to pull from later

clear temp a b toavg;
end

disp('Features averaged, now comparing');
Distance metric

nbrpts = featdepth; % number of distinct time points to find per tag
separablesplit = 100; % minimum distance (in samples) time points have to be separated by
edgecuts = 2000; % how many points to exclude on the edges

% Compare all tags to each other (get separation distances)
comb = nchoosek(1:nbrtags, 2); % all possible combinations
[a b] = size(comb);

meandiffmat = zeros(length(comb), datal); % Initialize
meandiff = zeros(nbrfeats, datal);

for featidx = 1:nbrfeats % loop through all features
    tempavg = squeeze(avgdata(featidx,:,:));
    for idx = 1:a
        meandiffmat(idx,:) = abs(tempavg(comb(idx,1),:) - tempavg(comb(idx,2),:));
    end
    meandiff(featidx,:) = mean(meandiffmat);
    clear tempavg;
end

% Find mean value of stdevs for use in finding max points with lowest stdev
stdevmax = squeeze(mean(stdev,2));
jointdiff = zeros(1,datal); % initialize
for idx = 1:28
    jointdiff(idx,:) = (meandiff(idx,:) ./ stdevmax(idx,:));
end

% Remove edges
meandiff(:,1:edgecuts) = 0; % don't use edges
meandiff(:,end-edgecuts:end) = 0;
stdevmax(:,1:edgecuts) = 0; % don't use edges
stdevmax(:,end-edgecuts:end) = 0;
jointdiff(:,1:edgecuts) = 0; % don't use edges
jointdiff(:,end-edgecuts:end) = 0;

% Now find where meandiff is high but stdevmax is low
selected_feats.timepts = zeros(nbrfeats,nbrpts); % preallocate
selected_feats.timeptspread = zeros(nbrfeats,nbrpts);
selected_feats.meandiff = zeros(nbrfeats,nbrpts);
selected_feats.stdevmax = zeros(nbrfeats,nbrpts);
selected_feats.jointdiff = zeros(nbrfeats,nbrpts);
selected_feats.normjointdiff = zeros(nbrfeats,nbrpts);
for idx = 1:nbrfeats;
    disp([' Selecting features ' num2str(idx) ' of ' num2str(nbrfeats)]);
    skip = 0;
    splitdat = 0;

    switch peakdetect
    case 'separate'
        % original method- find max of meandiff and min of stdevmax simultaneously
        maxpct = 0.8; % percentage of max(meandiff) -> lower value is less restrictive
        minpct = 0.1; % percentage of max(stdevmax) -> higher value is less restrictive
        while length(splitdat) < nbrpts % how many distinct time points we want to find
            maxcuts = find(meandiff(idx,:)>(maxpct*max(meandiff(idx,:))));
            mincuts = find(stdevmax(idx,:)<(minpct*max(stdevmax(idx,:))));
            comparepoints = intersect(maxcuts, mincuts); % find intersection
            splitdat = split_contiguous(comparepoints, separablesplit); % split intersection array into contiguous blocks
    end
end
if length(splitdat) < nbrpts & maxpct > 0.1 \% criterion were too strict
    - try reducing mean restriction first
    maxpct = maxpct - 0.06;
elseif length(splitdat) < nbrpts & minpct < 0.75 \% mean restriction
didn't cut it - loosen the stdev restriction
disp('Loosened stdev restriction');
    maxpct = 0.8;
    minpct = minpct + 0.05;
elseif length(splitdat) < nbrpts
    skip = 1;
    splitdat = cell(1,nbrpts);
    disp('Skipped');
end
end

switch joint
    case{'joint'}
        % create new array - meandiff/stdevmax, then find its peaks
        jointpct = 0.08; \% percentage of max(jointdiff) -> lower value is less restrictive
        while length(splitdat) < nbrpts \% how many distinct time points we want to find
            comparepoints = find(jointdiff(idx, :) > (jointpct * max(jointdiff(idx, :) ) ));
            splitdat = split_contiguous(comparepoints, separableSplit); \% split intersection array into contiguous blocks
            if length(splitdat) < nbrpts & jointpct > 0.1
                jointpct = jointpct - 0.02;
            elseif length(splitdat) < nbrpts \% should never happen
                skip = 1;
                splitdat = cell(1,nbrpts);
                disp('Skipped');
            end
        end
    end
end

if skip == 1
    for splitidx = 1:length(splitdat)
        temp = cell2mat(splitdat(splitidx));
        tempspread(splitidx) = temp(end) - temp(1); \% spread (in samples)
        temptpts(splitidx) = floor(mean(temp)); \% pick time point in middle (round down if necessary)
    end
end

switch peakdetect
    case{'separate'}
        % original method- find max of meandiff and min of stdevmax simultaneously
        [sortedstdevmax, sortorder] = sort(stdevmax(idx, temptpts)); \% sorts based on stdev
        % !! Change to both sort by jointdiff? !!
    case{'joint'}
        % create new array - meandiff/stdevmax, then find its peaks
        [sortedstdevmax, sortorder] = sort(jointdiff(idx, temptpts), 'descend'); \% sorts based on jointdiff
end

sortedtpts = temptpts(sortorder(1:nbrpts)); \% reorder sorted time points, also cut down to size (nbrpts)
selected_feats.timepts(idx, :) = sortedtpts; \% save everything to the structure
selected_feats.timeptsSpread(idx, :) = tempspread(sortorder(1:nbrpts));
selected_feats.meandiff(idx, :) = meandiff(idx, sortedtpts);
selected_feats.jointdiff(idx, :) = jointdiff(idx, sortedtpts);
selected_feats.normjointdiff(idx, :) = jointdiff(idx, sortedtpts) ./ max(jointdiff(idx, :));
selected_feats.stdevmax(idx, :) = sortedstdevmax(1:nbrpts);
clear tempspread temptpts sortedtpts sortorder sortedstdevmax

% Find overall top n features
[sorted_normjointdiff, si] = sort(selected_feats.normjointdiff(:, 'descend')); % Find
top n features, sorted by normalized jointdiff
[topfeats_iidx, topfeats_jidx] = ind2sub([nbrfeats, featdepth], si(1:topnfeats)); % get
indices of top feats
selected_feats.topfeats_indices = [topfeats_iidx topfeats_jidx]; % save indices to
structure
selected_feats.feats = feats; % save feature names to structure

if viewselected == 1; % view selected points
  for plotidx = 1:nbrfeats;
    switch peakdetect
    case{'separate'}
      plot(mean_diff(plotidx,:)); hold on; plot(stdev_max(plotidx,:),'r--');
      plot(selected_feats.timepts(plotidx,:), selected_feats.mean_diff(
          plotidx,:), 'ks', 'LineWidth', 3);
      hold off; legend('Mean difference', 'Max stdev', 'Selected points');
    pause;
    case{'joint'}
      plot(joint_diff(plotidx,:)); hold on; plot(selected_feats.timepts(
          plotidx,:), selected_feats.joint_diff(plotidx,:), 'ks', 'LineWidth', 3);
      hold off;
    pause;
    end
  end
end

% Pull data at top selected time points for all data
topfeats.timepts = selected_feats.timepts(si(1:topnfeats)); % timepoints to pull
for idx = 1:topnfeats
  for tagidx = 1:nbrtags
    featdata(tagidx, idx).data = allcomp(tagidx, topfeats_iidx(idx)).data(
      :,topfeats_timepts(idx)); % STRUCTURE is featdata(tagidx, topfeatsdata).data(
      nbrdatasets, nbrchs)
  end
end

% reshape into something more useful
for tagidx = 1:nbrtags
  tempshape = [featdata(tagidx,:).data];
  alldata(tagidx).data = reshape(tempshape,[],topnfeats); % STRUCTURE is
  alldata(nbrtags).data(nbrdatasets, topnfeats)
end
pulledfeatdata = vertcat(alldata.data); % pulledfeatdata(nbrtags*nbrdatasets, topnfeats)
topnfeats = dataset(pulledfeatdata,char(tagnames)); % save as array of prtools datasets

%%%% Radar subfunctions %%%%%
function radar_getRTI
% Analyzes data from coffee can radar to get range vs time intensity (RTI) plot
% In:
% Out:
% NOTE: set up-ramp sweep from 2-3.2V to stay within ISM band
% Change fstart and fstop below when in ISM band

%read the raw data .wave file here
[Y, FS, NBITS] = wavread('radar_2013011o-133749.wav');

% constants
\[ c = 3\times 10^8 \text{ m/s} \] speed of light

% Radar parameters
Tp = 20E-3; % (s) pulse time
N = Tp*FS; % # of samples per pulse
% fstart = 2250E6; % (Hz) LFM start frequency for example
% fstop = 2690E6; % (Hz) LFM stop frequency for example
% fstart = 2402E6; % (Hz) LFM start frequency for ISM band
% fstop = 2495E6; % (Hz) LFM stop frequency for ISM band
f = linspace(fstart, fstop, N/2); % instantaneous transmit frequency

% Range resolution
rr = c/(2*BW);
max_range = rr*N/2;

% The input appears to be inverted
trig = -1*Y(:,1);
s = -1*Y(:,2);
clear Y;

% Parse the data here by triggering off rising edge of sync pulse
count = 0;
thresh = 0;
start = (trig > thresh);
for ii = 100:size(start,1)
    if start(ii) == 1 & mean(start(ii-11:ii-1)) == 0
        % start2(ii) = 1;
        count = count + 1;
        sif(count,:) = s(ii:ii+N-1);
        time(count) = ii*l/FS;
    end
end

% Check to see if triggering works
% plot(trig,'b');
% hold on;
% plot(start2,'r');
% hold off;
% grid on;

% Subtract the average
ave = mean(sif,1);
for ii = 1:size(sif,1)
    sif(ii,:) = sif(ii,:) - ave;
end
zpad = 8*11/2;

% ATI plot
figure;
v = dbv(fft(sif,zpad,2));
S = v(:,1:size(v,2)/2);
m = max(max(S));
imagesc(linspace(0,max_range,zpad),time,S-m,[-80, 0]);
colorbar;
ylabel('time (s)');
xlabel('range (m)');
title('RTI without clutter rejection');

% Pulse cancelor RTI plot
figure;
sif2 = sif(2:size(sif,1,:),)-sif(1:size(sif,1)-1,:);
v = fft(sif2,zpad,2);
S=v;
R = linspace(0,max_range,zpad);
for ii = 1:size(S,1)
    S(ii,:) = S(ii,:).*R.*(3/2); %Optional: magnitude scale to range
end
S = dbv(S(:,1:size(v,2)/2));
m = max(max(S));
imagesc(R,time,S-m,[-80, 0]);
colorbar;
ylabel('time (s)');
xlabel('range (m)');
title('RTI with 2-pulse cancelor clutter rejection');

% X2 pulse mag only cancelor
% figure(30);
% clear v;
% for ii = 1:size(sif,1)-1
%     v1 = abs(ifft(sif(ii,:),zpad));
%     v2 = abs(ifft(sif(ii+1,:),zpad));
%     v(ii,:) = v2-v1;
% end
% S=v;
% R = linspace(0,max_range,zpad);
% for ii = 1:size(S,1)
%     S(ii,:) = S(ii,:).*R.*(3/2); %Optional: magnitude scale to range
% end
% S = dbv(S(:,1:size(v,2)/2));
% m = max(max(S));
% imagesc(R,time,S-m,[-20, 0]);
% colorbar;
% ylabel('time (s)');
% xlabel('range (m)');
title('RTI with 2-pulse mag only cancelor clutter rejection');

function out = dbv(in)
out = 20*log10(abs(in));

AFIT simulations
Create AFIT 3D input
% afit_create_3d_input.m
% Create input file of parameters for AFIT simulation on SciClone ONLY for 3d arb scatterers
% Designed to run on SciClone
% It's a good idea to test adding scatterer to space locally first ('import.stl.scatterer')
% Usage: 'afit.create.3d_input'
% In: NULL
% Out: NULL, writes 'in.file' and 'arbscatt.file' in current directory
% Dependencies: pulse input ('.mat') and correct STL file in working directory

% Eric A. Dieckman (V&H)
% 2 June 2010
% Last edited: 18 Sep 2013 EAD

% log
% 18 Sep 2013 EAD - STL and input .mat files are read from central location
% 19 Sep 2013 EAD - Changed input to 'import_stl_scatterer' to the (x,y,z) format
% Fixed error in writing sex to file (was writing ass)
% FOR NOW, stick to space sizes where x = y (z can be different)
% TODO: Input to scattering simulation should be (x,y,z) instead of (y,x,z)?
% change so that only 'isinside' is saved to 'arbscatt.file' and reassembled
inside Cpp code using scat placement

function afit_create_3d_input

%%%% USER PARAMS %%%

% for testing % testing = 1; % if true (1), override 3d input with simple sphere centered at scatterer placement

% path to director for input files
filedir = '/sciclona/home04/ead712/AFIT/devel/';
pulename = 'ARAinput.mat'; % 'longpulse.mat', 'shortpulse.mat' % audio input pulse
scatname = 'simple_car.stl'; % STL scatterer file

% Material and space parameters
den = 1.2; % material density of air - (kg/m^3)
soundspeed = 343;

fmax = 400; % max freq (determines ds)
maxt = 1650; % max time steps
outevery = 65; % output this many time steps

% Sinspace size (m)
spaces = 1; % x-dim (height)
spacy = 1; % y-dim (width)
spacex = 1; % z-dim (depth) -> propagation direction

% Scatterer placement (m)
scatz = 0.25; % z-dim (height)
scaty = spacex/2; % y-dim (width)
scatx = 1.5; % z-dim (depth)

% Calculate important settings for 'in.file' %
wavelength = soundspeed/fmax;
ds = wavelength/8 % spatial step size (m); at least 6 points per wavelength
dt = ds/(soundspeed*sqrt(3)); %time step size (s)
%dt = 1/(soundspeed*sqrt(3/ds^2)); %time step size (s)

ssy = makeeven(spacy/ds) % simulation space size in the y-dim (width)
ssx = makeeven(spacex/ds) % x-dim (height)
ssz = makeeven(spacez/ds) % z-dim (depth) -> propagation direction

input = data.pulse;
df(1:maxt) = 0;
df(1:length(input)) = input; % 'input' is the drive function

% Convert space from m to step sizes
scsy = makeeven(scaty/ds);
scsz = makeeven(scatz/ds);
scsx = 0; % NOT USED IN SIMULATION

% Parameters for scatterer in 'in.file' - most aren't used in this case
rftype = 3; % scatterer type
numscat = 1; % number of scatterers (1)
scatden = -1; % scatterer density (-1 for rigid)
scatc = -1; % scatterer sound speed (-1 for rigid)
rrad = ssy*ssz*(ssz+2); % total size

if testing == 1 % override 3D scatterer with simple sphere
    rftype = 0;
    rrad = makeeven(0.2/ds); % sphere radius
end

% Write 'in.file' - DO NOT CHANGE ORDER %
fp=fopen('in.file','w');
% this is the propagation direction (z for our coords)
fprintf(fp, ' X8.0f ', ssz);
fprintf(fp, ' X8.0f ', ssy);
fprintf(fp, ' X8.0f ', ssx);

fprintf(fp, ' X2.20f ', ds);
fprintf(fp, ' X2.20f ', dx);
fprintf(fp, ' X15.6f ', den);
fprintf(fp, ' X15.6f ', soundspeed);

fprintf(fp, ' X8.0f ', maxt);
fprintf(fp, ' X8.0f ', outevery);

fprintf(fp, ' Xl5.6f ', df(l:maxt));

fprintf(fp, ' X8.0f ', numscat);
fprintf(fp, ' X8.0f ', rtype);
fprintf(fp, ' X8.0f ', scsy);
fprintf(fp, ' X8.0f ', scsx);
fprintf(fp, ' X8.0f ', scsz);
fprintf(fp, ' X8.0f ', scs);
fprintf(fp, ' X15.6f ', rrad);
fprintf(fp, ' X15.6f ', scatden);
fprintf(fp, ' X15.6f ', scatc);

fclose(fp);
disp('in.file successfully written');

%----- Write 'arbescatt.file' - DO NOT CHANGE ORDER -----%
% Some warning of possible file size?

[arbescatt, isinside] = import_stl_scatterer([filedir scatname], ds, [scatx scaty scatz],
[spacex spacey spacez]); % create logical space

fp=fopen('arbescatt.file','w');
fprintf(fp, 'Xd ', arbscatt);
fclose(fp);
disp('arbescatt.file successfully written');

%----- Helper functions -----%
function evenified = makeeven(toround)
% Ensures given value is int even; if odd, adds 1 to make even
evenified = round(toround);
if mod(evenified,2) == 1 % needs to be even
    evenified = evenified + 1;
end

function [arbescatt, isinside] = import_stl_scatterer(filename, ds, objorigin, spacesize)
% Imports STL file and creates array of scattering boundary for use in AFIT sims
% stlread based on 'cad2matdemo'
% USAGE:
% Input: 'objorigin' - where to place scatterer origin (in m), 'spacesize' - size of
% simulation space (in m)
% Output:
% Dependencies: drawMesh (part of the geom3d package)
% Eric A. Dieckman (V&N)
% 26 August 2013
% Last edited: 26 August 2013 EAD
[faces, vertices, c] = stlread(filename); % import STL file - origin must be correct and
scale in mm
figure; drawMesh(vertices, faces); % draw mesh as a patch object
figure; scatter3(vertices(:,1), vertices(:,2), vertices(:,3)); % just the vertices
should give us enough info

transverts = vertices; % vertices translated to new origin (and y-corrected)
for idx = 1:3
    transverts(:,idx) = transverts(:,idx) + objorigin(idx)*1000; % move object origin
    (given in m)
    if max(transverts(:,idx)) > spacesize(idx)*1000 || min(transverts(:,idx)) < 0 % ensure
    all values are within the simspace
        disp('Scatterer placement not possible - outside of space');
        return;
    end
end

ssverts = vertices; % convert from mm scale to simspace scale (using ds)
for idx = 1:3
    ssverts(:,idx) = transverts(:,idx)/(ds*1000); % need to round?
end
figure; scatter3(ssverts(:,1), ssverts(:,2), ssverts(:,3)); drawMesh(ssverts,faces);

% Find points inside this object - to save memory, focus only on space where scatterer
is located
scatstart = floor(min(ssverts));
scatend = ceil(max(ssverts));
isinside = inpolyhedron(faces, ssverts, scatstart(1):1:scatend(1), scatstart(2):1:
scatend(2), scatstart(3):1:scatend(3), 'flipnormals', 'true');

% View results:
scatstart(3):1:scatend(3));
figure; hold on;
plot3(xgrid(isinside), ygrid(isinside), zgrid(isinside), 'bo', 'MarkerFaceColor', 'b');
plot3(xgrid(~isinside), ygrid(~isinside), zgrid(~isinside), 'ro'), axis image; hold off;

% Now create logical array for entire space as input to simulation - !! there's probably
a better way to do this !!
fullspace = [makeeven(spacesize(1)/ds), makeeven(spacesize(2)/ds), makeeven(
spacesize(3)/ds) + 2]; % size of full space (steps) (with 2 added)
arbscatt = logical(zeros(fullspace)); % this could get big quickly - use logical to
reduce size

% This is probably not necessary anymore (needed when confused (xyz) with (yxz))
if size(isinside,1) == scatend(1) - scatstart(1) + 1 % nothing got switched around
    arbscatt(scatstart(1):scatend(1),scatstart(2):scatend(2),scatstart(3):scatend(3)) =
    isinside;
elseif size(isinside,1) == scatend(2) - scatstart(2) + 1 % x and y got switched around
    arbscatt(scatstart(2):scatend(2),scatstart(1):scatend(1),scatstart(3):scatend(3)) =
    isinside;
else
    disp('Something wrong with logical array');
end

function [fout, vout, cout] = stlread(filename)
% Reads ASCII stl file and returns a vertex list and face list for Matlab patch command
fid=fopen(filename, 'r'); % Open the file, assumes STL ASCII format.
if fid == -1
    error('File could not be opened, check name or path.');
end
% STL files of form:
%solid BLOCK
% color 1.000 1.000 1.000
% facet
% normal 0.000000e+00 0.000000e+00 -1.000000e+00
% normal 0.000000e+00 0.000000e+00 -1.000000e+00
% normal 0.000000e+00 0.000000e+00 -1.000000e+00
% outer loop
% vertex 5.000000e-01 -5.000000e-01 -5.000000e-01
% vertex -5.000000e-01 -5.000000e-01 -5.000000e-01
% vertex -5.000000e-01 5.000000e-01 -5.000000e-01
% endloop
% endfacet
%
% The first line is object name, then comes multiple facet and vertex lines.
% A color specifier is next, followed by those faces of that color, until
% next color line.
%
% CAD_object_name = sscanf(fgetl(fid), ' X*s Xs'); % CAD object name, if needed.
% vnum=0; % Vertex number counter.
% report_num=0; % Report the status as we go.
% VColor = 0; %
% while feof(fid) == 0 % test for end of file, if not then do stuff
% tline = fgetl(fid); % reads a line of data from file.
% fword = sscanf(tline, ' Xs'); % make the line a character string
% % Check for color
% if strncmpi(fword, 'c',1) == 1; % Checking if a "C"olor line, as "CB" is 1st char.
% VColor = sscanf(tline, ' X*s Xf Xf Xf'); % & if a C, get the RGB color data of the
% face.
% end % Keep this color, until the next color is used.
% if strncmpi(fword, 'v',1) == 1; % Checking if a "V"ertex line, as "V" is 1st char.
% vnum = vnum + 1; % If a V we count the # of V's
% report_num = report_num + 1; % Report a counter, so long files show status
% if report_num > 249; %
% disp(sprintf(' Reading vertix num: %d.',vnum));
% report_num = 0; %
% end
% v(:,vnum) = sscanf(tline, ' X*s Xf Xf Xf'); % & if a V, get the XYZ data of it.
% c(:,vnum) = VColor; % A color for each vertex, which will color the
% faces.
% end % we "*s" skip the name "color" and get the data.
% Build face list; The vertices are in order, so just number them.
% % fnum = vnum/3; % Number of faces, vnum is number of vertices. STL is triangles.
% flist = 1:vnum; % Face list of vertices, all in order.
% F = reshape(flist, 3,fnum); % Make a "3 by fnum" matrix of face list data.
% % % Return the faces and vertexs.
% % fout = F'; % Orient the array for direct use in patch.
% vout = v'; %
% cout = c'; %
% fclose(fid);

function IN = inpolyhedron(varargin)
INPOLYHEDRON Tests if points are inside a 3D triangulated (faces/vertices) surface

% IN = INPOLYHEDRON(FV,QPTS) tests if the query points (QPTS) are inside the
% patch/surface/polyhedron defined by FV (a structure with fields 'vertices' and
% 'faces'). QPTS is an N-by-3 set of XYZ coordinates. IN is an N-by-1 logical
% vector which will be TRUE for each query point inside the surface.

246
INPOLYHEDRON takes faces/vertices separately, rather than in an FV structure.

IN = INPOLYHEDRON(..., XVEC, YVEC, ZVEC) allows for 3D gridded query points rather than an N-by-3 array of points. X, Y, and Z coordinates of the grid supplied in XVEC, YVEC, and ZVEC respectively. IN will return as a 3D logical volume with SIZE(IN) = [LENGTH(YVEC) LENGTH(XVEC) LENGTH(ZVEC)], equivalent to syntax used by MESHGRID. INPOLYHEDRON handles this input faster and with a lower memory footprint than using MESHGRID to make full X, Y, Z query points matrices.

INPOLYHEDRON(...,' PropertyName', VALUE,' PropertyName', VALUE,...) tests query points using the following optional property values:

- TOL - Tolerance on the tests for "inside" the surface. You can think of tol as the distance a point may possibly lie above/below the surface, and still be perceived as on the surface. Due to numerical rounding nothing can ever be done exactly here. Defaults to ZERO. Note that in the current implementation TOL only affects points lying above/below a surface triangle (in the Z-direction). Points coincident with a vertex in the XY plane are considered INside the surface. More formal rules can be implemented with input/feedback from users.

- GRIDSIZE - Internally, INPOLYHEDRON uses a divide-and-conquer algorithm to split all faces into a chessboard-like grid of GRIDSIZE-by-GRIDSIZE regions. Performance will be a tradeoff between a small GRIDSIZE (few iterations, more data per iteration) and a large GRIDSIZE (many iterations of small data calculations). The sweet-spot has been experimentally determined (on a win64 system) to be correlated with the number of faces/vertices. You can overwrite this automatically computed choice by specifying a GRIDSIZE parameter.

- FACENORMALS - By default, the normals to the FACE triangles are computed as the cross-product of the first two triangle edges. You may optionally specify face normals here.

- FLIPNORMALS - (Defaults FALSE). Triangle face normals are presumed to point towards the "inside" of the surface. If your surface normals are defined pointing "out" of the volume, set FLIPNORMALS to TRUE.

Example:

```matlab
% Empty voxel volume
tmpvol = zeros(20,20,20);
tmpvol(5:15,8:12,8:12) = 1;  % Turn some voxels on
tmpvol(8:12,5:15,8:12) = 1;  % Create the patch object
% Test SCATTERED query points
pts = rand(200,3)*12 + 4;  % Make some query points
in = inpolyhedron(fv, pts);  % Test which are inside the patch
% Display the result
patch(fv,'FaceColor','g','FaceAlpha',0.2)
plot3(pts(in,1),pts(in,2),pts(in,3),'bo','MarkerFaceColor','b')
plot3(pts(~in,1),pts(~in,2),pts(~in,3),'ro')
axis image

% Test STRUCTURED GRID of query points
gridLocs = 3:2:19;
[x,y,z] = meshgrid(gridLocs,gridLocs,gridLocs);
in = inpolyhedron(fv, gridLocs,gridLocs,gridLocs);
% Display the result
patch(fv,'FaceColor','g','FaceAlpha',0.2)
plot3(x(in),y(in),z(in),'bo','MarkerFaceColor','b')
plot3(x(~in),y(in),z(~in),'ro')
axis image
```

% TODO-list:
- Add IN/ON tolerance for in-plane edges (via user feedback)
- Improve overall memory footprint. (need examples with MEM errors)
- Implement an "ignore these" step to speed up calculations for:
  - Vertically oriented faces (no z-component in face normal)
  - Query points outside the convex hull of the faces/vertices input
% - Get a better/best gridSize calculation. User feedback?
% - Detect cases where X-rays or Y-rays would be better than Z-rays?

% Author: Sven Holcombe
% - 10 Jun 2012: Version 1.0
% - 28 Aug 2012: Version 1.1 - Speedup using accumarray
% - 07 Nov 2012: Version 2.0 - BEHAVIOUR CHANGE

**FACETS** is an unpacked arrangement of faces/vertices. It is [3-by-3-by-N],
with 3 l-by-3 XYZ coordinates of N faces.

```
[facets, qPts, options] = parseInputs(varargin{:});
numFaces = size(facets,3);
```

Function speed can be thought of as a function of grid size. A small number of grid
% squares means iterating over fewer regions (good) but with more faces/qPts to
% consider each time (bad). For any given mesh/queryPt configuration, there will be a
% sweet spot that minimizes computation time. There will also be a constraint from
% memory available - low grid sizes means considering many queryPt/faces at once,
% which will require a larger memory footprint. Here we will let the user specify
% gridsize directly, or we will estimate the optimum size based on prior testing.

```
if isempty(options.gridsize)
    gridSize = options.gridsize;
else
    gridSize = round((-1.0e-8*numFaces^2 + 0.00095*numFaces + 18));
    if numFaces>50000, gridSize = 40; end
end
```

**X** Find candidate qPts -> triangles pairs
% We have a large set of query points. For each query point, find potential
% triangles that would be pierced by vertical rays through the qPt. First,
% a simple filter by XY bounding box

```
% Calculate the bounding box of each facet
minFacetCoord = permute(min(facets(:,1:2,:)),[3 2 1]);
maxFacetCoord = permute(max(facets(:,1:2,:)),[3 2 1]);

% Set rescale values to rescale all vertices between 0(-eps) and 1(+eps)
scalingOffsets = min(minFacetCoord,...
    scalingRange = max(maxFacetCoord,...
% Based on scaled min/max facet coords, get the [lowX lowY highX highY] "grid" index
% of all faces
lovToHighGridIdx = floor(bsxfun(@divide, ...
    % Use min/max coordinates of each facet (+/- the tolerance)
    [minFacetCoord - eps, maxFacetCoord + eps] = scalingOffsets;
    [scalingRange + scalingOffsets] = scalingRange + 1;

% Build a grid of cells. In each cell, place the facet indices that encroach into
% that grid region. Similarly, each query point will be assigned to a grid region.
% Note that query points will be assigned only one grid region, facets can cover many
% regions. Furthermore, we will add a tolerance to facet region assignment to ensure
% a query point will be compared to facets even if it falls only on the edge of a
% facet's bounding box, rather than inside it.
    cells = cell(gridSize);
    [uniqLHgrids,uu,facetIdxs] = unique(lovToHighGridIdx,'rows');
```

```
tmpInds = accumarray(facetIdxs,1:length(facetIdxs),[1,0(z)(x)]);

248
for xi = 1:gridSize
    xyMinMask = xi > unqLHgrids(:,:1) & xi <= unqLHgrids(:,:3);
    for yi = 1:gridSize
        cells(yi,xi) = cat(1,tmpInds{xyMinMask & yi} > unqLHgrids(:,2) & yi <= unqLHgrids(:,4));
        % The above line (with accumarray) is faster with equiv results than:
        %  % cells(yi,xi) = find(ismember(facetInds, xyInds));
    end
    % With large number of facets, memory may be important:
clear lowToHighGridIdxs LHgrids facetInds tmpInds xyMinMask minFacetCoords maxFacetCoords
end

% Precompute the 3d normals to all facets (triangles). Do this via the cross product
% of the first edge vector with the second. Normalise the result.
allEdgeVecs = facets([2 3 1],:,:) - facets([1 3 2],:,:);
if isempty(options.facenormals)
    allFacetNormals = bsxfun(@times, allEdgeVecs(1,[2 3 1],:,:), allEdgeVecs(2,[3 1 2],:,:))
        - bsxfun(@times, allEdgeVecs(2,[2 3 1],:,:), allEdgeVecs(1,[3 1 2],:,:));
    allFacetNormals = bsxfun(@rdivide, allFacetNormals, sqrt(sum(allFacetNormals.^2,2)));
else
    allFacetNormals = permute(options.facenormals,[3 2 1]);
end
if options.flipnormals
    allFacetNormals = -allFacetNormals;
end
% Precompute the 2d unit vectors making up each facet's edges in the XY plane.
allEdgeUVecs = bsxfun(@rdivide, allEdgeVecs(:,1:2,:), sqrt(sum(allEdgeVecs(:,1:2,:).^2,2)));
% Precompute the inner product between edgeA.edgeC, edgeB.edgeA, edgeC.edgeB
allEdgeEdgeDotPs = sum(allEdgeUVecs .* -allEdgeUVecs([3 1 2],:,:),2) - 1e-9;

% Since query points are most likely given as a (3D) grid of query locations, we only
% need to consider the unique XY locations when asking which facets a vertical ray
% through an XY location would pierce.

% Gather the unique XY query locations
if ~options.griddedInput
    % Scattered query points were provided
    [uniqQpts,uu.unqQptInds] = unique(qPts(:,1:2),'rows');
    uniqQptCell = accumarray(uniqQptInds,1:length(uniqQptInds), [1,0(x)(x)]);
    qPtsViaUnqInds = @(ind)qPts(uniqQptCell(ind),:);
    outPxIndsViaUnqIndexMask = @(ind,mask)uniqQptCell(ind)(mask);
    outputSize = [size(qPts,1),1];
else
    % Structured query points were provided.
    [xmat,ymat] = meshgrid(qPts(:,1:2));
    uniqQpts = [xmat(:) ymat(:)];
    % A standard set of Z locations will be shifted around by different
    % uniqQpts XY coordinates.
    zCoords = qPts(3,:) * [0 0 1];
    qPtsViaUnqInds = @(ind)bsxfun(@plus, zCoords, [uniqQpts(ind,:) 0]);
    % From a given indice and mask, we will turn on/off the IN points under
    % that indice based on the mask. The easiest calculation is to setup
    % the IN matrix as a numZpts-by-numUnqPts mask. At the end, we must
    % unpack/reshape this 2D mask to a full 3D logical mask
    numZpts = size(zCoords,1);
    baseZinds = 1:numZpts;
    outPxIndsViaUnqIndexMask = @(ind,mask)(ind-1)*numZpts + baseZinds(mask);
    outputSize = [numZpts, size(oneUnqQpts,1)];
end
% Assign each query location to a grid region
unqQgridXY = floor(bsxfun(@div, bsxfun(@minus, unqQpts, scalingOffsetsXY),...
    scalingRangeXY) * gridSize) + 1;

% We are about to iterate over grid regions. Since some (relatively small) number of
% unique XY query points will belong to the same grid region, we want to find the
% changes in grid locations as we go through the unique XY query locations. Build
% that list.
newlnds = cat(1, 0, find(any(diff(unqQgridXY,[],1),2)), size(unqQgridXY,1));

% To fit nicely into the below calculations, we're going to reshape the query points
% from an N-by-2 array to a 1-by-2-by-1-by-N array. This will make it easier to do
% some tricky bsxfun() calls.
unqQpts = reshape(unqQpts',1,2,1,[]);

% Start with every query point NOT inside the polyhedron. We will iteratively find
% those query points that ARE inside.
IN = false(outputSize);
for i = 1:length(newlnds)-1
    fromTo = newlnds(i)+1:newlnds(i+1);

    % Gather information about this GRID. We need to know the grid indices, and from
    % that we get the facet indices of all triangles that enter this grid cell
    gridNoXY = unqQgridXY(fromTo(1),:);
    if any(gridNoXY>gridSize | gridNoXY<1), continue; end
    allFacetInds = cells(gridNoXY(2),gridNoXY(1));

    % If there are no facets in this grid region to consider, we need go no further
    if isempty(allFacetInds), continue; end

    % We get all the facet coordinates (ie, triangle vertices) of triangles that
    % intrude into this grid location. The size is [3-by-2-by-N_for the
    % [3-vertices-by-XY-by-N-triangles]
candVerts = facets(:,1:2,:allFacetInds);

    % We need to know about the query points. To check, for intersections with
    % triangles, we only need the distinct XY coordinates of the (possibly many)
    % query points.
    queryPtsXY = unqQpts(fromTo(1,:),fromTo);

    % Get unit vectors pointing from each triangle vertex to my query point(s)
    vert2ptVecs = bsxfun(@minus, queryPtsXY, candVerts);
    vert2ptUVecs = bsxfun(@div, vert2ptVecs, sqrt(sum(vert2ptVecs.^2,2)));

    % Get inner products between each edge unit vec and the UVs from qPt to vertex
    edgeQPntDotPs = sum(bsxfun(@times, edgeUVs, vert2ptVecs),2);
    qPntEdgeDotPs = sum(bsxfun(@times, edgeUVs, -vert2ptVecs([3 1 2],:,:)),2);
    % If both inner products 2 edges to the query point are greater than the inner
    % product between the two edges themselves, the query point is between the V
    % shape made by the two edges. If this is true for all 3 edge pair, the query
    % point is inside the triangle.
    resultIN = all(bsxfun(@gt, edgeQPntDotPs, edgeEdgeDotPs)) & bsxfun(@gt, qPntEdgeDotPs, edgeEdgeDotPs),1);
    resultONVERTEX = any(any(isnan(vert2ptUVecs),2),1);
    result = resultIN | resultONVERTEX;

    qPntHitsTriangles = any(result,3);
    % If NONE of the query points pierce ANY triangles, we can skip forward
    if ~any(qPntHitsTriangles), continue, end

    % In the next step, we'll need to know the indices of ALL the query points at
    % each of the distinct XY coordinates. Let's get their indices into "qPts" as a
% call of length M, where M is the number of unique XY points we had found.

for ptNo = find(qPtHitxTriangles); % Which of the unique "2D" points are we querying?
    unqQptInd = fromTo(ptNo);
    % Which facets does it pierce?
piercedFacetInds = allFacetInds(result(1,1,:,:ptNo));
    % Get the 1-by-3-by-N set of triangle normals that this qPt pierces
    piercedTriNorms = allFacetNormals(:, :, piercedFacetInds);
    % Pick the first vertex as the "origin" of a plane through the facet. Get the % vectors from each query point to each facet origin
    facetToQptVectors = bsxfun(@minus, qPtsViaUnqIndices(unqQptInd), facets(1,:,:,
piercedFacetInds));
    % Calculate how far you need to go up/down to pierce the facet's plane. % Positive direction means "inside" the facet, negative direction means % outside.
    facetToQptDists = bsxfun(@rdivide, ...
        sum(bsxfun(@times, piercedTriNorms, facetToQptVectors), 2), ...
        abs(piercedTriNorms(:, 3, :)));
    % Since it's possible for two triangles sharing the same vertex to % be the same distance away, I want to sum up all the distances % of triangles that are closest to the query point.
    absFacetDists = abs(facetToQptDists);
    closestFacetDists = sum(bsxfun(@times, facetToQptDists, bsxfun(@eq, absFaceDist, min(absFacetDists,[],3))),3);
    IN(outPxIndsViaUnqIndiceMask(unqQptInd, closestFacetDists>-options.tol)) = true;
end

% If they provided X,Y,Z vectors of query points, our output is currently a % 2D mask and must be reshaped to [LEN(Y) LEN(X) LEN(Z)].
if options.griddedInput
    IN = reshape(IN', cellfun(@numel, qPts([2 1 3])));
end

% Input handling subfunctions
function [facets, qPts, options] = parseInputs(varargin)
% Gather FACES and VERTICES
if isstruct(varargin{l}); % inpolyhedron(FVstruct, ...)
    if ~all(isfield(varargin{l},{'vertices','faces'}))
        error( 'Structure FV must have "faces" and "vertices" fields' );
    end
    faces = varargin{l}.faces;
    vertices = varargin{l}.vertices;
    varargin{l} = [] ; % Chomp off the faces/vertices
else % inpolyhedron(FACES, VERTICES, ...)
    faces = varargin{l};
    vertices = varargin{2};
    varargin{1:2} = [] ; % Chomp off the faces/vertices
end
% Unpack the faces/vertices into [3-by-3-by-N] facets. It's better to % perform this now and have FACETS only in memory in the main program, % rather than FACETS, FACES and VERTICES
facets = vertices;

251
facets = permute(reshape(facets(:, faces'), 3, 3), [2 1 3]);

% Extract query points
if length(varargin)<2 || ischar(varargin{2})
    % inpolyhedron(F, V, [x(:) y(:) z(:)], ...)
    qPts = varargin{1};
    varargin{1} = [];
    % Chomp off the query points
elseif varargin{1} == []
    % inpolyhedron(F, V, xVec, yVec, zVec, ...)
    qPts = varargin{1:3};
    % Chomp off the query points and tell the world that it's gridded input.
    varargin{1:3} = [];
    varargin = [varargin {'griddedInput',true}];
end

% Extract configurable options
options = parseOptions(varargin);

function options = parseOptions(varargin)
IP = inputParser;
IP.addParamValue('grids',
    varargin{2}, @(x)isnumeric(x) && ismatrix(x))
IP.addParamValue('tol', 0, @(x)isnumeric(x) && ismatrix(x))
IP.addParamValue('tol_ang', 1e-5, @(x)isnumeric(x) && ismatrix(x))
IP.addParamValue('tol_normals', [],
    @(x)isnumeric(x) && ismatrix(x));
IP.addParamValue('griddedInput',false);
IP.parse(varargin{:});
options = IP.Results;

/* afit.cpp
   AFIT simulation - development copy
   Eric A. Dieckman (WM)
   19 August 2013
   Last edited: 27 September 2013 EAD
   
   log
   02 Oct 2012 EAD - Changed from ascii output to visit compatible output
   19 Aug 2013 EAD - Forked from 'play.cpp', clean up
   Uses 'space.h' instead of 'acousticrect.h'
   (Eventually, don't use custom arrays)
   21 Aug 2013 EAD - Resolved error with saving data (misplaced delete)
   Added option to save data as vtk - NOT WORKING
   (Current solution uses bash script to read data into Visit as BOV format)
   12 Sep 2013 EAD - Added reading in 3d scatterer data (changes to 'space.h'
   for scatterer type 3)
   13 Sep 2013 EAD - Rename dimensions to reflect (y, x, z) geometry
   [(width, height, depth), propagation in +z direction]
   28 Sep 2013 EAD - Add savesinglepressure - just set outputevery to 1 in
   'afit_create_3d_input'
   *
   TODO: stop using custom arrays*/

#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>
#include <math.h>
#include "space.h"
//include "visit.writer.h"

using namespace std;

void master();
void slave();
void DistributeSimulationParameters();
void dump3Dascii(int t);
void dump3Dbin(int t);
//void dump3Dvtk(int t);

int rank, numworkers;
int maxr, outputevery, maxl;

//int whohaseline = 0;
int recordalineat = 1; // z-point to save pressure data (x and y points set to middle of sim space) - should be less than div

int main(int argc, char *argv[]) // initialize MPI
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numworkers); /* get number of nodes */

    if (rank == 0)
    master();
    else
    slave();

    MPI_Finalize();
    return 0;
}

// Master node! -- Distributes simulation space and receives data for output
// Master node is online! 

void master() {
    time_t start,end;
    time (&start);

    MPI_Status status;
    cout << "Master node is online! \n";

    DistributeSimulationParameters(); // Initialize each node

    ofstream pfile("pressuredata.ascii", ios::out);
    double al=0;

    for (int t=0; t<maxt; t++) {
        //MPI_Recv(&al, 1, MPI_DOUBLE, whohaseline, 858, MPI.COMM.WORLD, &status);
        //outFile << al << " ";
        //cout << "Time in Master node is: " << t << endl;

        if (outputevery == 1) { // Save pressure data at specific time point
            MPI_Recv(&al, 1, MPI_DOUBLE, 1, 858, MPI.COMM_WORLD, &status);
            // pulls from 1st node - recordalineat must be < div
            pfile << al << " ";
            cout << "Saved single pressure measurement at time: " << t << "\n";
        }

        if (t%outputevery == 0 && outputevery != 1) {
            //dump3Dascii(t);
            //cout << "Saved pressure data as ASCII at time: " << t << "\n";
            dump3Dbin(t);
        }
    }
    return 0;
}
cout << "Saved pressure data as binary at time: " << t << "\n";
    //dump3Dvtk(t);
cout << "Saved pressure data as vtk at time: " << t << "\n";
}
}
pfile.close();

time(&end);
printf("Total Run Time: \%.2f seconds\n", difftime(end,start));
return;

// Slave node! -- Does the grunt work
//---------------------------------------------------------------
void slave() {
    // --- Receive sim parameters from master and initialize ---
    MPI_Status status;
    MPI_Request request[2];

double simparams[10];
    MPI_Recv(simparams, 10, MPI.DOUBLE, 0, 201, MPI.COMM.WORLD, &status);

    space simspace;
simspace.num1 = simparams[0]+2; // number nodes in z-direction
    simspace.num2 = simparams[1]; // number of nodes in y-direction
    simspace.num3 = simparams[2]; // number of nodes in x-direction
    simspace.ds = simparams[3]; // spatial step size (m)
    simspace.dt = simparams[4]; // time step size (s)
    simspace.den = simparams[5]; // density
    maxt = simparams[6]; // total number of time steps
    outputevery = simparams[7]; // output every this many time steps
    simspace.xbeg = simparams[8]; // simspace x-starting position
    simspace.cc = simparams[9]; // sound speed

    int mm3 = simspace.num2*simspace.num3; // max size of y»x dim

    if (rank == 1) // node is on left
        simspace.type = 1;
    else if (rank == numvorkers) // node is on right
        simspace.type = 3;
    else
        simspace.type = 2; // node is in middle

    simspace.Init();

    // --- Receive drive function ---
    if (rank==1) {
        double *drive = new double[maxt];
        MPI_Recv(&drive[0], maxt, MPI.DOUBLE, 0, 202, MPI.COMM.WORLD, &status);
        simspace.df = drive;
    }

    // --- Receive reflector parameters ---
    int nr;
    double *rpars = new double[8];
    MPI_Recv(&nr, 1, MPI.INT, 0, 203, MPI.COMM.WORLD, &status);

    for (int i = 0; i < nr; i++) {
        MPI_Recv(rpars[0], 8, MPI.DOUBLE, 0, 204, MPI.COMM.WORLD, &status);
        simspace.addReflector(rpars[0], rpars[1], rpars[2], rpars[3], rpars[4], rpars[5],
            rpars[6], rpars[7]);
    }

    254
// --- Run simulation ---
//double al;
for (int t = 0; t < maxt; t++) {
    if (rank == 1)
        cout << "time: " << t << " " << simspace.num1 << " " << simspace.num2 << " " << simspace.num3 << " end!
    simspace.time = t;
    // if ((recordalineat >= simspace.zbeg) && (recordalineat < (simspace.zbeg + simspace.num-1))){
    //    al = ar.pp.val(recordalineat-ar.zbeg,100,100);
    //    MPI_Isend(kal, 1, MPI.DOUBLE, 0, 858, MPI.COMM.WORLD, request);
    //}
    if (outputevery == 1) {
        double al = simspace.pp.val(recordalineat,simspace.num2/2,simspace.num3/2);
        // AT WHICH POINT TO SAVE PRESSURE DATA (Z,Y,X)?
        MPI_Isend(&al, 1, MPI.DOUBLE, 0, 858, MPI.COMM.WORLD, request);
    }
    if (t%outputevery == 0 && outputevery != 1) { // sends output to master node
        //tosend = ar.pp;
        int len = simspace.pp.GetEvenVolLen(simspace.zbeg); // start at zbeg for each node
        double* x = simspace.pp.GetEvenVol(simspace.zbeg);
        MPI_Isend(&x[0], len, MPI.DOUBLE, 0, 1101, MPI.COMM.WORLD, request);
        delete [] x;
    }
    // --- Update Ps ---
    simspace.UpdatePs(1,1); // update left boundary
    simspace.UpdatePs(2,simspace.num-2); // update right boundary
    if (rank > 1) { // send left
        MPI_Isend(&simspace.pp.a[m2m3], m2m3, MPI.DOUBLE, (rank-1), 301, MPI.COMM_WORLD, request);
    }
    if (rank < numworkers) { // receive from right
        MPI_Recv(&simspace.pp.a[(simspace.num1-1)*m2m3], m2m3, MPI.DOUBLE, (rank+1),
        301, MPI.COMM_WORLD, &status);
    }
    // --- Update Vs ---
    simspace.UpdateVs(1,simspace.num-3);
    simspace.UpdateVs(simspace.num-2,simspace.num1-2);
    if (rank < numworkers) {
        MPI_Isend(&simspace.vl.a[(simspace.num1-2)*m2m3], m2m3, MPI.DOUBLE, (rank+1),
        302, MPI.COMM_WORLD, request);
    }
    if (rank > 1) {
        MPI_Recv(&simspace.vl.a[0], m2m3, MPI.DOUBLE, (rank-1), 302, MPI.COMM_WORLD, &status);
    }
    simspace.doDriveFunction();
}

// Reads in parameter file (in.file), distributes to all workers,
// and divides up the simulation space
// ===========================================================================
void DistributeSimulationParameters() {
    char inputFilename[] = "in.file";
    ifstream inFile;
    inFile.open("in.file", ios::in);
    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }

    double *simparams = new double[10];
    inFile >> simparams[0]; // maxi
    inFile >> simparams[1]; // max2
    inFile >> simparams[2]; // max3
    inFile >> simparams[3]; // ds
    inFile >> simparams[4]; // dt
    inFile >> simparams[5]; // default den
    inFile >> simparams[6]; // default speed of sound
    inFile >> simparams[7]; // maxt
    inFile >> simparams[8]; // output every
    maxt = simparams[6];
    outputevery = simparams[7];
    // maxi = simparams[0];
    *
    /* // DISPLAY VALUES
    cout<<"Total z is"<<totalz;
    cout<<"maxi is: "<<simparams[0]"\n";
    cout<<"max2 is: "<<simparams[1]"\n";
    cout<<"max3 is: "<<simparams[2]"\n";
    cout<<"ds is: "<<simparams[3]"\n";
    cout<<"dt is: "<<simparams[4]"\n";
    cout<<"den is: "<<simparams[5]"\n";
    cout<<"c is: "<<simparams[6]"\n";
    cout<<"maxt is: "<<simparams[7]"\n";
    cout<<"outputevery is: "<<simparams[8]"\n";
    */
    // Send initial data to each node
    int div, divaccum = 0;
    for (int n = 1; n < numvorkers; n++) {
        div = (maxi/(numvorkers));
        if ((n-1) <= (max(0,numvorkers)))
            div++;
            /* divide space along xi direction */
        simparams[0] = div;
        cout<<"Divided space (div) is "<<div"\n";
        simparams[8] = divaccum; // tells the worker where its starting x location is
        cout<<"Worker's starting locations (divaccum) is "<<divaccum"\n";
        MPI_Send(fcsimparams[0], 10, MPI.DOUBLE, n, 201, MPI.COMM.WORLD);
        divaccum = divaccum+div;
        //if ((whohasaline==0)&&(divaccum==recordalineat))
        // whohasaline=n;
    }
    cout << "whohasaline = " << whohasaline << "\n";
    // --- Read in drive function and send to worker number 1 ---
    double *drive = new double[maxt];
    for (int i = 0; i<maxt; i++) {
        inFile >> drive[i];
    }
    MPI_Send(&drive[0], maxt, MPI.DOUBLE, 1, 202, MPI.COMM.WORLD);
    // --- Read in reflectors and distribute to all workers ---
}

256
int numref;
inFile >> numref;
double *rpars = new double[8];
cout << " Number of reflectors: " << numref << endl;
for (int n = 1; n <= numworkers; n++) {
    MPI_Send(numref, 1, MPI_INT, n, 203, MPI_COMM_WORLD);
}

for (int i = 0; i < numref; i++) {
inFile >> rpar[0]; // reflector type
inFile >> rpar[1]; // reflector position in x1
inFile >> rpar[2]; // reflector position in x2
inFile >> rpar[3]; // reflector position in x3 - (start for cylinder)
inFile >> rpar[4]; // reflector position in x3 - (end for cylinder)
inFile >> rpar[5]; // reflector radius
inFile >> rpar[6]; // reflector density
inFile >> rpar[7]; // reflector speed of sound

    /* // DISPLAY VALUES
    cout << "reflector type is: " << rpar[0] << "\n";
    cout << "reflector position x1: " << rpar[1] << "\n";
    cout << "reflector x2: " << rpar[2] << "\n";
    cout << "reflector x3 (start for cylinder): " << rpar[3] << "\n";
    cout << "reflector x3 (end for cyl): " << rpar[4] << "\n";
    cout << "reflector radius is: " << rpar[5] << "\n";
    cout << "reflector density is: " << rpar[6] << "\n";
    cout << "reflector speed of sound is: " << rpar[7] << "\n";
*/
}

MPI.Send(rpar[0], 8, MPI.DOUBLE, n, 204, MPI.COMM.WORLD);

inFile.close();
return;

void dump3Dascii(int t) // pressure data as ascii{
    MPI_Status status;
    double *data3d;
    int len;

    stringstream strm;
    strm << t;
    string fname = "data3d_at_t_" + strm.str() + ".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    MPI_Recv(tlen, 1, MPI.INT, n, 1101, MPI.COMM.WORLD, kstatus);
    if (n == 1) data3d = new double[len];
    
    MPI_Recv(data3d[0], len, MPI.DOUBLE, n, 1102, MPI.COMM.WORLD, &status);
    for (int i = 0; i < len; i++)
        outFile << data3d[i] << " "; // old way - writes ASCII files
}
delete [] data3d;
void dump3Dbin(int t) // pressure data as binary
{
    MPI_Status status;
    double *data3d;
    int len;

    stringstream strm;
    strm << t;
    string fname = "data3d_at_t_" + strm.str() + "bin";
    ofstream outFile(fname.c_str(), ios::binary);

    outFile << ar.num1-2 << " " << ar.num2 << " " << ar.num3 << " ";
    for (int i3 = 0; i3 < ar.num3; i3++)
        for (int i2 = 0; i2 < ar.num2; i2++)
            for (int i1 = 0; i1 < ar.num1-1; i1++)
                outFile << ar.pp.val(i1, i2, i3) << " ";
    outFile << ar.num1-2 << " " << ar.num2 << " " << ar.num3 << " ";
    for (int n = 1; n < numworkers; n++)
    {
        MPI_Recv(tlen, 1, MPI.INT, n, 1101, MPI.COMM_WORLD, &status);
        if (n==1) data3d = new double[len];
        MPI_Recv(&data3d[0], len, MPI.DOUBLE, n, 1102, MPI.COMM_WORLD, &status);

        for (int i = 0; i < len; i++)
            outFile.write((char *) &data3d[i], sizeof(data3d[i]));
    }
    delete [] data3d;
    outFile.close();
    return;
}

/*
void dump3Dvtk(int t) // pressure data as vtk (ascii) !! can do binary?
{
    MPI_Status status;
    double *data3d;
    int len;

    stringstream strm;
    strm << t;
    string fname = "data3d_at_t_" + strm.str() + "vtk";

    for (int n = 1; n < numworkers; n++)
    {
        MPI_Recv(tlen, 1, MPI.INT, n, 1101, MPI.COMM_WORLD, &status);
        if (n==1) data3d = new double[len];
        MPI_Recv(&data3d[0], len, MPI.DOUBLE, n, 1102, MPI.COMM_WORLD, &status);

        write_regular_mesh(fname, 0, (94, 94, 94), 1, len, 0, "Pressure", vars);
    }
    delete [] data3d;
    return;
}
*/

/* space.h
* Geometries for AFIT simulation - development copy
*/
#include <iostream>
#include "array3D.h"
#include "array3D_int.h"
// #include "transducer.h"

#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))

class space {
public:
    space() {}
    ~space() {}
    
    int num1;       // number of grid points in z direction
    int num2;       // number of grid points in y direction
    int num3;       // number of grid points in x direction
    int totals;     // MAY BE ABLE TO REMOVE
    int abc;        // number of abc points on each end
    double ds;      // spatial step size (meters)
    double dt;      // time step size (seconds)
    double den;     // density (kg/m^3)
    double cc;      // default speed of sound
    int zbeg;       // z start position (where divacuum ends)
    int type;       // type -> 1 = left , 2 = middle, 3 = right

    array3D v1;     // 1 - velocities
    array3D v2;     // 2 - velocities
    array3D v3;     // 3 - velocities
    array3D pp;     // pressures
    array3D c;      // speed of sound
    array3D d;      // density

    array3D_int B;  // Boundary Array
    int time;       //
    double *df;    // drive function

private:
    double dtods;
    int 11,12,13;

public:
    void Init()
    {
        v1.Init(num1,num2,num3);
        v2.Init(num1,num2,num3);
        v3.Init(num1,num2,num3);
        pp.Init(num1,num2,num3);
        c.Init(num1,num2,num3,cc);
        d.Init(num1,num2,num3,den);
    
}
B.Init(num1, num2, num3, type);

dtods = dt/ds;

time = 0;

abc = 40;

void UpdatePs(int zs, int zend)
{
    for (i1 = zs; i1 <= zend; i1++)
        for (i2 = 1; i2 < num2; i2++)
            
            pp.setindz(i1, i2, 1); vi.setindz(i1, i2, l); v2.setindz(i1, i2, 1);
            v3.setindz(i1, i2, 1); d.setindz(i1, i2, 1);
            for (i3 = 1; i3 < num3; i3++)
                { 
                    pp.ev( pp.v()-dtoda*d.v()*(v1.v()-(v1.v()-(v1.v()-(v2.v()-(v3.v()-(v3.v()-(v3.v())") ));
                    pp.incindz(); vi.incindz(); v2.incindz(); v3.incindz(); d.incindz();
                }

}

void UpdateVs(int zs, int zend)
{
    for (i1 = zs; i1 <= zend; i1++)
        for (i2 = 1; i2 < num2; i2++)
            
            pp.setindz(i1, i2, 1); vi.setindz(i1, i2, l); v2.setindz(i1, i2, 1);
            v3.setindz(i1, i2, 1); d.setindz(i1, i2, 1);
            for (i3 = 1; i3 < num3; i3++)
                { 
                    v1.sv( v1.v() - 2*dtods/(d.v()+d.vlp())*(pp.vlp()-pp.v()) );
                    if (i2<num2-l) { v2.sv( v2.v() - 2*dtods/(d.v()+d.v2p()) *(pp.v2p()-pp.v()) );
                          if (i3<num3-l) { v3.sv( v3.v() - 2*dtods/(d.v()+d.v3p())*(pp.v3p()-pp.v()) );
                          pp.incindz(); vi.incindz(); v2.incindz(); v3.incindz(); d.incindz();
                    }

}

// Plane Boundary Conditions
// ---------------------------------------------
for (i1 = zs; i1 <= zend; i1++)
{
    for (i2 = 0; i2 < num2; i2++)
        
        v3.set(i1, i2, 0, v3.val(i1, i2, 1));
        v3.set(i1, i2, num3-1, v3.val(i1, i2, num3-2));
    }

for (i3 = 0; i3 < num3; i3++)
{
    v2.set(i1, 0, i3, v2.val(i1, i1, 13));
    v2.set(i1, num2-1, i3, v2.val(i1, num2-2, i3));
}

// ---------------------------------------------

// Rigid Reflectors!
for (il = zs; il <= zend; il++)
for (i2 = 0; i2 < num2-1; i2++)
{
    B.setindx(il,i2,0); vl.setindx(il,i2,0); v2.setindx(il,i2,0);
    v3.setindx(il,i2,0);
    for (i3 = 0; i3 < num3-1; i3++)
    {
        if (B.v(0) == 2)
        {
            if (B.v2p == 2) vl.sv(0);
            if (B.v3p == 2) v2.sv(0);
            if (B.v3p == 2) v3.sv(0);
        }
        B.incindx(); vl.incindx(); v2.incindx(); v3.incindx();
    }
}

// Comment these lines out, KR 10/27/09
// doABCsO; // absorbing bcs
// doBackABCs(totals2);

void doABCsO()
{
    int sabc = 25;
    double per;
    for (il = 2; il < num1-2; il++)
    for (i2 = 0; i2 < num2; i2++)
    for (i3 = 1; i3 < aabc; i3++)
    {
        per = (1-.002*(aabc-i3));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-i3-1); v2.setindx(il,i2,num3-i3-1); v3.setindx(il,i2,num3-i3-1);
        v1.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,num2-i2-1,i3); v2.setindx(il,num2-i2-1,i3); v3.setindx(il,num2-i2-1,i3);
        v1.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        per = (1-.002*(aabc-i2));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-i2-1,i3); v2.setindx(il,i2,num3-i2-1,i3); v3.setindx(il,i2,num3-i2-1,i3);
        v1.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
    }
}

void doBackABCsO(int TotalZ) // ABC on the backside of the space (maxi) {
    int sabc = 25;
    double per;
    for (il = max(TotalZ-aabc-1,zbeg); (il >= zbeg) & (il < (zbeg+num1-1)); il++)
    {
        per = (1-.002*(aabc-il));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-il-1); v2.setindx(il,i2,num3-il-1); v3.setindx(il,i2,num3-il-1);
        per = (1-.002*(aabc-il));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-il-1); v2.setindx(il,i2,num3-il-1); v3.setindx(il,i2,num3-il-1);
        per = (1-.002*(aabc-il));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-il-1); v2.setindx(il,i2,num3-il-1); v3.setindx(il,i2,num3-il-1);
        per = (1-.002*(aabc-il));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-il-1); v2.setindx(il,i2,num3-il-1); v3.setindx(il,i2,num3-il-1);
        per = (1-.002*(aabc-il));
        vl.setindx(il,i2,i3); v2.setindx(il,i2,i3); v3.setindx(il,i2,i3);
        v1.setindx(il,i2,num3-il-1); v2.setindx(il,i2,num3-il-1); v3.setindx(il,i2,num3-il-1);
    }
}
void doDriveFunction()
{
    if (type == 0)
    {
        v1.setindx(0,0,0);
        for (i2 = 0; i2 < num2; i2++)
            for (i3 = 0; i3 < num3; i3++)
                v1.sv(v1.v() - 2*dtods/(d.val(1,i2,i3)+d.val(0,i2,i3))*(pp.val(1,i2,i3)-pp.val(0,i2,i3)+df[time]));
        v1.incindx();
    }
}

void addReflector(double typ, double p1, double p2, int start3, int end3, double rad, double dd, double rc)
{
    if (typ == 0) //sphere
    {
        for (i1 = 0; i1 < num1; i1++)
            for (i2 = 0; i2 < num2; i2++)
                for (i3 = 0; i3 < num3; i3++)
                    if (((i1+zbeg-l-pl)*(i1+zbeg-l-pl) + (i2-p2)*(i2-p2) + (i3-start3)*(i3-start3)) < rad*rad)
                        if ((rc == -1) && (dd == -1))
                            { B.set(i1,i2,i3,2); }
                        else
                            { c.set(i1,i2,i3,rc); d.set(i1,i2,i3,dd); }
    }
    else if (typ == 1) //cylinder
    {
        for (i1 = 0; i1 < num1; i1++)
            for (i2 = 0; i2 < num2; i2++)
                if (((i1+zbeg-l-pl)*(i1+zbeg-l-pl) + (i2-p2)*(i2-p2)) < rad*rad)
                    if ((rc == -1) && (dd == -1))
                        { B.set(i1,i2,i3,2); }
                    else
                        { c.set(i1,i2,i3,rc); d.set(i1,i2,i3,dd); }
    }
}
else if (typ == 2) //rectangle
{
    for (i1 = 0; i1 < num1; i1++)
        for (i2 = 0; i2 < num2; i2++)
            for (i3 = 0; i3 < num3; i3++)
                if (((i1+zbeg-1 >= start3) && (i1+zbeg-1 <= end3))
                    if ((rc == -1) && (dd == -1))
                    { B.set(i1,i2,i3,2);
                    }
                else
                {
                    c.set(i1,i2,i3,rc);
                    d.set(i1,i2,i3,dd);
                }
}
else if (typ == 3) //arb 3d scatterer from STL file
{
    char inputFilename[] = "arbscatt.file"
    ifstream inFile;
    inFile.open("arbscatt.file", ios::in); //arbscatt file follows (y,x,z) orientation
    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }
    int tss = rad; // total size of space
    int *scatterspace = new int[tss];
    for (int i = 0; i<tss; i++) {
        inFile >> scatterspace[i]; // read in entire space including scatterer
    }
    // Nested loops through width, height, depth to compute array indices for linear array
    // here, num1 is z, num2 is y, and num3 is x -> (y,x,z) == (num2, num3, num1)
    // for (i2 = 0; i2 < num2; i2++) // width (num2/12)
    // for (i3 = 0; i3 < num3; i3++) // height (num3/i3)
    // for (i1 = 0; i1 < num1; i1++) { //depth (num1/11)
    // if (scatterspace[(long)i2*(long)num3*(long)num1 + (long)i3*(long)num1 + (long)i1] == 1)
    // for (i1 = 0; i1 < num1; i1++)
    // for (i2 = 0; i2 < num2; i2++) // height (num3/13)
    // for (i3 = 0; i3 < num3; i3++) { //depth (num1/11)
    // if (scatterspace[(i1+zbeg-1)*num2 + num3 + i2] == 1)
    // { if ((rc == -1) && (dd == -1))
    //     { B.set(i1,i2,i3,2);
    //     }
    //     else
    //     {
    //         c.set(i1,i2,i3,rc);
    //         d.set(i1,i2,i3,dd);
    //     }
    //     //i++;
    // } } in.File.close();
}
OpenSCAD parametric vehicle model

// parametric_vehicle_model.scad
// Parametric model of vehicle fronts
// For AFIT scattering simulations
//
// Eric A. Dieckman (WAN)
// 18 September 2013
// Last edited: 18 Sep 2013 EAD

// ------ USER INPUT ------
fronthoodheight = 1275; // x
backhoodheight = 1276; // x
windshieldheight = 1530; // x
grilloffset = 1; // difference between top of grill and bottom of grill (z)
hoodlength = 1; // z
windshielddepth = 50;
tireoffset = 1880; // distance between tires and front/rear of vehicle (z)
wrads = 500; // wheel radius
wthick = 300; // wheel thickness (y)
bodywidth = 2600; // y
bodylength = 3000; // Total length of vehicle for sim (z)
// For boxtruck (comment out otherwise)
//boxtruckdepth = 1500; // where box starts (z)
//boxtruckheight = 750; // height of box (x)
//boxtruckoverlap = 100; // width overlap on each side (y)
// ------ END USER INPUT ------

// BUILD UP MODEL
hoodoffset = backhoodheight - fronthoodheight;
translate([wrads, -bodywidth/2,0]) // to correct origin
// Wheels:
translate([0, wthick, wrads + tireoffset]) rotate([90, 0, 0])
cylinder(h=wthick, r=wrads);
// pass front
translate([0, bodywidth, wrads + tireoffset]) rotate([90, 0, 0])
cylinder(h=wthick, r=wrads); // driver front
//translate([0, wthick, bodylength - (wrads + tireoffset)]) rotate([90, 0, 0])
cylinder(h=wthick, r=wrads); // pass rear
//translate([0, bodywidth, bodylength - (wrads + tireoffset)]) rotate([90, 0, 0])
cylinder(h=wthick, r=wrads); // driver rear

// Hood and Windshield:
translate([0, bodywidth/2,0]) inclinedplane(fronthoodheight, bodywidth, grilloffset); // grill tilt
translate([0, 0, grilloffset]) cube(size = [fronthoodheight, bodywidth, bodylength]); // engine compartment

translate([fronthoodheight, bodywidth/2, grilloffset]) inclinedplane(hoodoffset, bodywidth, hoodlength); // angle top of hood
translate([backhoodheight, bodywidth/2, grilloffset+hoodlength]) inclinedplane(windshieldheight, bodywidth, windshielddepth); // windshield
translate([fronthoodheight, 0, grilloffset+hoodlength]) cube(size = [hoodoffset, bodywidth, bodylength-hoodlength]); // fill in passenger compartment
translate([backhoodheight, 0, grilloffset+hoodlength+windshielddepth]) cube(264
size = [windshieldheight, bodywidth, bodylength-hoodlength-windshielddepth]);
// passenger compartment

// For boxtruck:
// translate([0, -boxtruckoverlap, boxtruckdepth]) cube(size = [backhoodheight+windshieldheight+boxtruckheight, bodywidth+2*boxtruckoverlap, bodylength-boxtruckdepth+grilloffset])); // fill in passenger compartment

module Inclinedplane(height, width, length) // x, y, z
{
  hw = width/2;
  polyhedron (points = [[0, -hw, 0], [0, hw, 0], [0, hw, length], [0, -hw, length],
    [height, -hw, length], [height, hw, length]], triangles = [[0,3,2], [0,2,1], [3,0,4],
    [1,2,6], [0,5,4], [0,1,5], [5,2,4], [4,2,3], ]);}

265
Bibliography


267


272


275


Vita

Eric A. Dieckman was born in Independence, Missouri on 3 May 1986. Upon graduation from high school in 2004, he attended Truman State University in Kirksville, Missouri. Here Eric conducted research in musical and architectural acoustics, receiving a Bachelor of Science degree in Physics in May 2008. For the next year, Eric studied Architectural Acoustics at Rensselaer Polytechnic Institute in Troy, New York, receiving a Master of Science degree in May 2009. From 2009 through 2013 he continued his graduate studies in the Applied Science Department at The College of William and Mary in Williamsburg, Virginia, working as a Graduate Research Assistant in the Nondestructive Evaluation and Robotics laboratory. Eric currently resides in New Haven, Connecticut. In his free time, Eric enjoys backpacking, swing dancing, and being a volunteer firefighter/EMT.