

Multivariable Process Diagnostics for Plasma Systems

A thesis submitted for the degree of Doctor of Philosophy.

By

David Coates M.Eng.Sc.

School of Electronic Engineering

Dublin City University

Research Supervisors:

Professor Patrick McNally,

Professor Miles Turner

Submitted November 2020.



This page intentionally blank

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy (PhD) is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:  (Candidate) ID No.: 80001589 Date: 23-Nov-2020

This page intentionally blank

Acknowledgements

First, I would like to express my thanks to my supervisor who started me on this course. I thank Dr. Stephen Daniels sincerely for guiding me as I considered the work involved in the undertaking of PhD studies. He introduced me to the exciting world of Plasma systems and I am most grateful. With his departure to greater heights, I then transferred to a new supervisor. I would also like to express my sincerest gratitude to my PhD supervisor Professor Patrick McNally for the continuous support of my PhD study. He had the patience to provide the motivation and the knowledge to guide me when I needed it. There were a quite a few moments when he urged me to press on and follow through. I am most privileged to have such a patient, knowledgeable supervisor and mentor for my PhD study.

Besides my direct advisor, I would like to thank Professor Miles Turner, Dr. Noel Murphy and Dr. Prince Anandarajah for their guidance, comments and encouragement. They presented me with the challenges to ensure that my research was thorough from a number of differing perspectives. To all my colleagues for their generosity and kind words in assisting me in this.

I would like to thank my wife Ann for her patience, her assistance and most importantly her support in this and all matters.

This page intentionally blank

Table of Contents

Chapter 1 Research Motivation	21
1.1. Multi Variable, Scalable Sensing Systems	24
1.2. Research Topics	28
1.2.1. Frequency-Resolved and Time-Resolved Sensors on Oxford Instruments PlasmaLab 100 Capacitively Coupled Plasma Tool.....	28
1.2.2. Event Classification. Processing at the Edge.....	31
1.2.3. Scalable Data Acquisition in an Industrial Production Environment.....	32
1.3. Summary.....	33
Chapter 2 Near and Far Radio Frequency Fields of Capacitively Coupled Plasma 37	
2.1. Introduction	37
2.2. CCP Machines Under Examination	42
2.2.1. Oxford 100 CCP Brief Description.....	43
2.2.1. Novellus Concept 2 Brief Description.....	44
2.3. RF Operation and the Sheath.....	44
2.4. Remote Sensing of Low-Pressure Plasma in Radio Near Field.....	46
2.5. Sheath Formation	58
2.6. Harmonic Generation	61
2.7. Summary.....	63
Chapter 3 Software Defined Spectral Analysis for B- and E-Field Plasma Analysis 65	
3.1. Introduction	65
3.2. Radio Spectrum Analysis: Non-Invasive Plasma Monitoring.	65
3.3. Design Requirement of Non-Invasive E- and B-Field Analyzer	67
3.4. Software Defined Radio System	68
3.5. Commercial Off-The-Shelf (COTS) Based System Components.....	70
3.5.1. Brief Description of the SignalHound SA44B Spectrum Analyzer	70
3.5.2. Spike Software for the SignalHound SA44B.....	72
3.5.3. BBR and High-Speed Acquisition	72
3.6. B-Field Antenna Subsystem	73
3.7. Design of Loop Antenna	77
3.7.1. Mutual Inductance.....	79
3.8. Received Power and Current within the Plasma.....	86
3.9. Plasma Voltage and Current for Oxford Instruments PlasmaLab 100 CCP. .	89
3.10. Balanced Differential Antenna for Plasma Chamber.....	90
3.11. Companding (Compression/Expansion) Harmonic Signal System.....	94
Chapter 4 Arc Detection using Time Domain Analysis	97
4.1. Introduction	97
4.2. Time Domain Detectors	97
4.2.1. Square Law Envelope Detector	98
4.2.2. Hilbert Transform Detector.....	100
4.2.3. Wavelet Transform Detector.....	101
4.3. Oxford Instruments PlasmaLab 100 CCP Tool Arcing.....	106
4.4. Simple Threshold Detector for Oxford 100 Arcing	107

4.5. Square Law Envelope Detector for Arc Detection.....	112
4.6. Hilbert Transform Detection of Arcing Events	115
4.7. Wavelet Transform Detection of Arcing Events	124
4.7.1. Continuous Wavelet Transform Detection of Arcing Events	125
4.7.2. Discrete Wavelet Transform for Detection of Arcing Events.....	127
4.7.3. DWT Cascaded Filter Design	129
4.7.4. Reconstruction of Signal from Wavelet Transform	131
4.8. Plasma Arc Event Detection Techniques	136
4.9. Summary.....	137
Chapter 5 Simulation Models of Capacitively Coupled Plasma	139
5.1. Introduction	139
5.2. Non-Linear Plasma Modelling.....	142
5.3. Capture of B-field Harmonics in a Capacitively Coupled Plasma System...	148
5.4. Revised Equivalent Circuit and Spice Simulation.	152
5.5. Hot and Cold Electron Jets Simulation	161
5.6. Improved model for Geometric Resonance and Hot & Cold Electron Injection	166
5.7. Tool Side Monitoring Models	176
Chapter 6 K-Nearest Neighbour & Radial Basis Function Modelling for Spectral Analysis.....	177
6.1. Introduction	177
6.2. K-Nearest Neighbour (KNN) Operation.....	178
6.3. Radial Basis Function (RBF) Network / Operation	179
6.4. Software Implementation of KNN Engine.	181
6.4.1. Oxford Instruments PlasmaLab 100 KNN Modelling Engine.....	183
6.4.2. Experimental Data Set.....	189
6.4.3. KNN for Pressure (Single Linear Mean Model).....	203
6.4.4. Segmented KNN for Gas-Flow Rates	204
6.4.5. Boosted KNN for gas flow prediction at different pressures	209
6.5. Principal Component Analysis of Pressure (and Gas-Flow)	213
6.6. Harmonic Recording Optimization	216
6.7. Summary.....	217
Chapter 7 Laplace Transform Modelling of Oxford Instruments PlasmaLab 100 CCP 219	
7.1. Introduction	219
7.2. Basic Transfer Function	220
7.3. Oxford Instruments PlasmaLab100 Laplace Transfer Function	222
7.4. Oxford Instruments PlasmaLab 100 CCP System Modelling.....	225
7.5. S-Domain Transfer Solutions	229
7.6. Match Unit and the Transfer Function.....	233
7.7. Summary.....	238
Chapter 8 Distributed Scalable Multivariable Industrial Data Acquisition	239
8.1. Introduction	239
8.2. Architecture at the Edge - Scalable Data Collection	241
8.3. Analytics Bus for the Edge.....	243
8.3.1. Homogeneous Interface	243
8.3.2. Federated Architecture.....	244
8.3.3. Locality	244
8.3.4. Administrative Boundaries	244
8.3.5. Secure Routing	244

8.3.6. Publish-Subscribe Paradigm	244
8.3.7. Incremental Deployment.....	245
8.4. Existing Platforms	245
8.5. Internet of Things (IoT) Rich Compute Edge Platforms	247
8.6. Operation of a MQ system.....	248
8.7. Network Scaling at the IOT Level	251
8.8. Sensor Protocol for Scalable Data Acquisition.....	255
8.8.1. Scalable Hierarchical Acquisition Protocol (SHAPer).....	256
8.8.2. Description of Base Components SHAPer.....	257
8.8.3. Sections of SHAPer	257
8.9. Hierarchical / Tree Database	263
8.10. Summary	266
Chapter 9 Future Work Beyond PhD Studies	267
9.1. Introduction	267
9.2. Development of Non-Invasive Sensors	267
9.3. Pseudo 3D Planar Optical Observation.	268
9.4. Spectral 2D Observation.	268
9.5. Phase-Locked Pseudo 3D Observation.	268
References.....	273
Appendix A Microwave Plasma Excitation with Autotuner	279
A.1 Introduction	279
A.2 ECR Powered Plasma	280
A.2.1 Daihen Microwave Autotuner	282
A.1.2.2 Daihen Automatic Microwave Tuner	288
Appendix B BBR format Tool.....	301
Appendix C TekTronix RSA306 format Tool.....	306
Appendix D Ocean Optics Acquisition Subsystem for Windows®	313
Appendix E Radio Frequency Spectrometer Acquisition for Windows® ..	328
Appendix F SquareLaw Demodulator.....	343
Appendix G Hilbert Transform Demodulator.....	345
Appendix H Continuous and Discrete Wavelet Demodulator.....	347
Appendix I Software Pattern Matching Engine (PME) Implementation and Results	349
Appendix J Software For Environmental Monitoring of Plasma Systems..	354
Appendix K Software for Analog and Digital Acquisition for Monitoring of Plasma Systems	362
Appendix L Software for Monitoring the Optical Spectrum of Plasma Systems operating on Linux®	380
Appendix M Software for Monitoring the Magnetron Radio Emission Spectrum of Plasma Systems under Linux®	392
Appendix N Software for Monitoring the Daihen Autotuner on the Hitachi 712 ECR Plasma System operating on Linux®	399
Appendix O Recording of Daihen Measurements.....	418

Appendix P Novellus Concept 2 DC Plasma Deposition Syetem and FRES	422
Appendix Q Bespoke SDR system design for FRES and TRES	425
Q.1 Introduction	425
Q.1.1 FPGA Mezzanine Cards (FMC)	427
Q.1.2 Operating the FMCOMMS3 / AD9361 / Downconverter Acquisition System	430
Q.1.3 Software Build for Analog Devices IIO Oscilloscope as Development Tool for SDR	430
Q.1.4 Loopback testing of FMCOMMS3 / AD9361 and software displays	432
Q.2 Time Domain Radio Emission Spectra, Redesign for Zero IF	436
Appendix R KNN Modelling Software	437
Appendix S Laplace Transform Modelling Software	446
Appendix T Laplace Transform Video Pressure Transfer Functions	455
Appendix U Netlist for Geometric Resonance and Hot & Cold Electron Injection	456

Table of Figures

Figure 1 Hierarchical Sensor and Analyses-at-Edge Topology	27
Figure 2 Capacitively Coupled Plasma (CCP)	40
Figure 3 Equivalent Circuit of Capacitively Coupled Plasma	41
Figure 4 Oxford Instruments PlasmaLab 100 CCP Plasma Chamber with FRES Antenna Location	43
Figure 5 Near Field Magnetic Coupling (Kelly & McNally, 2017) (<i>In their original concept there was no direct or indirect coupling to the powered electrode</i>) ...	47
Figure 6 General Overview of FRES Acquisition.....	48
Figure 7 Screen Shot of FRES Acquisition Using SA44B Spike Software (reproduced by special permission)	49
Figure 8 Distribution of Secondary Electrons Emitted by Silver According to Energy of Impinging Electron (Rudberg, 1930) (reproduced from (Chapman, 1980)).	51
Figure 9 Capacitively Coupled Plasma, dc Blocking	52
Figure 10 Sheath Potential Asymmetric Electrode at Positive and Negative Parts of RF cycle (May, 1997).....	53
Figure 11 Unequal Electrodes, with ' <i>Floating Potential</i> ' and Time-Averaged Potential.....	54
Figure 12 Variation of Potential around a Perturbation	55
Figure 13 Sheath Formation at Floating Substrate.....	58
Figure 14 Variation in Ion and Electron Density with Potential $V(x)$ in a Sheath, reproduced from (Chapman, 1980).	60
Figure 15 Fourier Transform of B-dot Signals (Paul A Miller, 2006)	61
Figure 16 Phase Relationship for the First Three Harmonics across the Plasma (Paul A Miller, 2006)	62
Figure 17 Basic Block Diagram of Digital Radio System.....	68
Figure 18 Initial SDR Non-Invasive FRES design.....	69
Figure 19 : Predicted Impedance for Atmospheric Plasma with Constant Power Density 50W/ cm³ (Ge & Zhang)	74
Figure 20 Variation of the Wavefield Impedance with Distance from the Source (Morgan, 2007)	75
Figure 21 PCB B-Field Antenna Construction.....	78
Figure 22 Generic Types of Magnetic Field Probe (R Ediss, 2018)	79
Figure 23 Beehive 100C Loop Antenna Calibration Certificate	81
Figure 24 Loop Antennae Frequency Response (dBm vs MHz) (Beehive Electronics Inc., 2019)	83
Figure 25 KF40 port with Quartz Glass and Location for Beehive 100C Antenna	84
Figure 26 Location of Bespoke Plain-coil Antenna for FRES High-order Harmonic Acquisition	85
Figure 27 Location of Custom Plain-coil Antenna Flush to the Inside Chamber Wall for FRES high-order Harmonic Acquisition	86
Figure 28 Operation of Oxford Instruments PlasmaLab 100 CCP Tool with Ar Plasma with Al Foil to Initiate Arcing.....	89
Figure 29 Recording of E- and B-field on Oxford Instruments PlasmaLab 100 CCP Tool using Rigol 500 MHz Oscilloscope	90

Figure 30 Differential Balun Isolated Cabling System.....	91
Figure 31 Differential Balun System Implementation.....	92
Figure 32 Bespoke Differential B-field Antenna in RF Shielded Chamber Enclosure (Dimensions of metal enclosure box = 110 mm X 120 mm X 120 mm).....	93
Figure 33 Characteristic Harmonic Signal Level for Oxford Instruments PlasmaLab 100 CCP Tool.....	95
Figure 34 Companding Transfer Function and Subsequent Acquired Signal	96
Figure 35 Basic Wavelets for Transforms (Andrew Tan, 2016).....	102
Figure 36 Morse Wavelet with Different Time-Bandwidths (Matlab, 2018)	103
Figure 37 Foil Used to Initiate Arcing Events with High Pressure in the Oxford Instruments PlasmaLab 100 CPP Tool Chamber	107
Figure 38 Screenshot of Real time E- and B-field Acquisition on the Oxford Instruments PlasmaLab 100 CCP - Forced Arcing. The B-field signal is shown in blue; the E-field signal is shown in yellow.....	108
Figure 39 Recorded E- and B-fields. Oxford Instruments PlasmaLab 100 CCP Forced Arcing with a Single Event at 1300 ns.....	109
Figure 40 High Order Low Pass Filter Normalized Response for LPF Detector.	109
Figure 41 Arcing Event with LPF Response on TRES E- & B-Fields for Arc Detection (Primary 13.56 MHz)	111
Figure 42 Low Pass Filter Normalized Response (10 MHz cutoff, Order 400, F_s = 500 MHz) for Square-Law Envelope Detector	113
Figure 43 Square Law Detector, blue=(E-field, B-field), red=Squarelaw (B-field, E-field)	114
Figure 44 Relative Fourier Transforms of the B-Field Acquisition before Arcing Event (blue) and across Arcing Event (red).	116
Figure 45 Relative Fourier Transforms of the E-Field Acquisition before Arcing Event (blue) and across Arcing Event (red).	117
Figure 46 Hilbert Transform of B-field, Real Component of HT and Instantaneous Phase Angle of HT of the B-field	119
Figure 47 Hilbert Transform of E-field, Real Component of HT and Instantaneous Phase Angle of HT of the E-field	120
Figure 48 Hilbert 3D Plot Perspective #1, showing the Phase and Amplitude of the Hilbert Transform of the B-field.	122
Figure 49 Hilbert 3D plot Perspective #2 showing the Phase and Amplitude of the Hilbert Transform of the B-field	123
Figure 50 Plot of CWT of the B-field with Log Scale Frequency Bands with Confidence Curve.....	126
Figure 51 Plot of CWT of the E-field with Log Scale Frequency Bands with Confidence Curve.....	126
Figure 52 B-field Short Window Spectrogram Arc Event (at time 1.350 μ s)	128
Figure 53 E-field Short Window Spectrogram Arc Event (at time 1.350 μ s)	128
Figure 54 Discrete Wavelet Transform Data Flow using HPF and LPF	130
Figure 55 Reconstructed and Original B-field Signal from Inverse CWT (almost indistinguishable)	132
Figure 56 Inverse CWT B-Field Showing the Reconstruction Deviation at Higher Frequencies	134
Figure 57 Reconstructed and Original E-Field Signal from Inverse CWT.....	135

Figure 58 Limit cycles of unforced Van der Pol oscillator representing a limit cycle system.....	140
Figure 59 Van der Pol oscillator in time domain with varying values of μ (Vaclav Satek, 2019 (Sept))	141
Figure 60 Generator, Matching Unit, Cabling and Non-Linear Plasma Equivalent Circuit (Frederik Schmidt T. M., 2018)	142
Figure 61 Voltage and Current Using the Global Plasma Model (Frederik Schmidt J. T., 2019).....	143
Figure 62 Asymmetric Electrode Plasma Chamber with Location of Flush Antenna.....	144
Figure 63 Hot Energetic Beams of Electrons During RF Cycle (Sebastian Wilczek, 2016).....	147
Figure 64 Antenna Flush to Oxford Instruments PlasmaLab 100 CC Plasma Chamber Wall	150
Figure 65 Harmonic B-Field Spectrum 20 MHz to 1 GHz (power scale is dBm)	151
Figure 66 Capacitively Coupled Plasma Equivalent Circuit Model with Varactor.....	153
Figure 67 13.559 MHz Simulated Equivalent Circuit with Tunable Varactor CCP Time-Series.....	154
Figure 68 FFT of CC Plasma Varactor Equivalent Circuit 10 MHz to 250 MHz V(ANT).....	155
Figure 69 FFT of CC Plasma Varactor Equivalent Circuit 10 MHz to 350 MHz I(R1).....	155
Figure 70 RES Measured B-field Response Curve (Kelly, McNally), ...	156
Figure 71 Simulated Harmonic Content Equivalent Circuit.....	157
Figure 72 Tuned Harmonic Equivalent Circuit for CCP	158
Figure 73 Time Response Simulation for Tunable Varactor CCP Equivalent Circuit for Plasma Bulk Current (I(R1)).....	159
Figure 74 FFT of Tunable Equivalent Circuit of CCP for Bulk Plasma Current... ..	160
Figure 75 Simulated Harmonic Content Tuned Equivalent Circuit for Bulk Plasma Current (comparison to untuned circuit model)	161
Figure 76 Equivalent Circuit of CCP with Provision for 'Reverse' Hot Electrons.....	162
Figure 77 B-field and E-field High-Speed Acquisition using Agilent 40 Gs scope for TRES	163
Figure 78 Equivalent Plasma Circuit Time-Resolved with 'Hot Reverse' Electrons.....	163
Figure 79 Simulated FFT Output of Plasma Equivalent Circuit with 'Hot Reverse' Electrons.....	164
Figure 80 Comparison of Normalized Harmonics for Equivalent Circuit with Tuned Elements and Reverse Phase Currents for Hot & Cold Electron Beams.....	164
Figure 81 Comparison of Measured Harmonics and Simulated Harmonics with Varactor, Tuned Elements and Single Shot Phased Hot and Cold 'Reverse' Currents	165
Figure 82 Geometric Resonance with Hot & Cold Electron Injection Simulation Overview.....	167
Figure 83 Normalized 8 pole Chebyshev Bandpass Filter for Geometric Resonance Simulation (Red is gain plot and Blue is phase plot).....	168

Figure 84 Geometric Resonance with Hot & Cold Electron Injection Circuit.....	170
Figure 85 Spectrum Response from Hot and Cold Electron Injection Model with Characteristic Geometric Resonance	172
Figure 86 Eight Pole Geometric Resonance Simulation Test.....	173
Figure 87 Geometric Resonance Harmonic Output with Simulated Plasma Harmonic Drive	174
Figure 88 Simulated Hot and Cold Electron Beams with Characteristic Geometric Resonance Mapping	175
Figure 89 RBF TRES Event Neural Network Data Flow	180
Figure 90 Screenshot Capture of Collection of FRES Harmonics with Plasma Recipe Variation.....	184
Figure 91 Antenna Location for FRES acquisition for High Harmonic Content .	186
Figure 92 Oxford Instruments PlasmaLab 100 Tool RF Power Supply Unit	187
Figure 93 Adjacent RF PSU Frequencies on Two Chambers of Oxford Instruments PlasmaLab100 CCP Tool (screenshot taken from Spike [™] software).	188
Figure 94 Recording of Example FRES Harmonic (50th of 13.5596 MHz) at 677.98 MHz.....	188
Figure 95 Plot of RES Harmonics as a Function of Varying Chamber Pressure .	190
Figure 96 Plot of RES Harmonics as a Function of Gas Flow Change	190
Figure 97 Harmonic Planes for 20, 25 and 30 sccm 02 , respectively.....	194
Figure 98 Box Car Plots for the Three Gas-flow Regimes of 20, 25 and 30 sccm, respectively.....	199
Figure 99 Comparison of First Three Principal Component Loadings for 20, 25, 30 sccm Gas-flows, respectively	202
Figure 100 KNN Model Prediction Accuracy for Gas-flow Rates	204
Figure 101 Variation of Harmonics as a Function of Gas-flow for Different Pressures	206
Figure 102 First Principal Coefficient Variance of Harmonics for 20, 30, 40, and 50 millitorr Changes.....	208
Figure 103 Experimental Measurement Variation for 20 millitorr Chamber Pressure.	210
Figure 104 Experimental Measurement Variation for 30 millitorr Chamber Pressure	211
Figure 105 Experimental Measurement Variation for 40 millitorr Chamber Pressure	211
Figure 106 Experimental Measurement Variation for 50 millitorr Chamber Pressure	211
Figure 107 KNN Model Prediction Accuracy for Gas-flow Rates Accounting for Experimental Measurement Variation.....	212
Figure 108 Harmonic Principal Component Contribution for Pressure and Gas- flow Measurements	214
Figure 109 First Four Principal Component Planes for Plasma Pressure and Gas- flow FRES Measurements on Oxford Instruments PlasmaLab100 CCP Tool	215
Figure 110 Transfer Function of Oxford Instruments PlasmaLab 100 CCP System.....	220
Figure 111 Basic Circuitry for Laplace Transform Modelling of Oxford Instruments PlasmaLab 100 CCP System	222
Figure 112 Riley Unified Model Schematic (Paul A Miller, 2006).....	224

Figure 113 Time Domain Transfer Function (Chamber Pressure = 20 millitorr)	227
Figure 114 Time Domain Transfer Function (Chamber Pressure = 30 millitorr)	227
Figure 115 Time Domain Transfer Function (Chamber Pressure = 40 millitorr)	228
Figure 116 Time Domain Transfer Function (Chamber Pressure = 50 millitorr)	228
Figure 117 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 20 millitorr	230
Figure 118 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 30 millitorr	231
Figure 119 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 40 millitorr	231
Figure 120 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 50 millitorr	232
Figure 121 Impedance Matching Unit Basic Equivalent Circuit	234
Figure 122 Oxford Instruments PlasmaLab 100 CCP Plasma Transfer Function Excluding the Calculations for Match Unit	236
Figure 123 Oxford Instruments PlasmaLab 100 CCP Plasma Transfer Function Including the Calculations for Match Unit	237
Figure 124 Basic MQ System Structure	249
Figure 125 Example Routing Exchange Based on Rabbit MQ ®	250
Figure 126 Hierarchical Data Flow for IOT System	254
Figure 127 Hitachi 712 ECR Plasma with Daihen Stub Autotuner	281
Figure 128 Example 3 stub manual tuner	282
Figure 129 Electron Cyclotron Hitachi 712 Energy Flow.	283
Figure 130 Power Transfer for Source vs Load Impedance.....	284
Figure 131 Daihen SMA-20B Autotuner Mechanics.....	286
Figure 132 Daihen WR430 Autotuner, Controller and Acquisition Box	289
Figure 133 Mapping of the Voltage to Current at Load.....	290
Figure 134 Run 1, Stub Recorded positions for Two-step Wafer Etch Recipe run	292
Figure 135 Run 1, Impedance and Phase Correction for Two-step Wafer Etch Recipe Run	292
Figure 136 Normalized Admittance Smith Chart Stub Operation for 3 Stubs	294
Figure 137 Run 1, Vload Reflection Coefficient Matching on Daihen Autotuner	295
Figure 138 Run 1, Vload Reflection Coefficient Matching on Daihen Autotuner	295
Figure 139 Smith Chart Plot of Daihen Autotuner Correction	296
Figure 140 Run 2, Forward Power as Measured by Daihen Autotuner Antenna	297
Figure 141 Run 2, Daihen Stub 1, 2 and 3 Recorded Positions (into the Waveguide).....	298
Figure 142 Run 2, The Corresponding Vload Matching by the Daihen Autotuner	299
Figure 143 Run2, Vload as Recorded from the Daihen Autotuner	299
Figure 144 BBR Processing Flow Diagram	301

Figure 145 Novellus Concept 2 with FRES and OES.....	422
Figure 146 Novellus Concept 2 FRES reception	423
Figure 147 Novellus Concept 2 OES signal	424
Figure 148 Block Diagram Mixer to ADC / DAC of Analog Devices AD9361	425
Figure 149 Block Diagram AD9631 SDR with Downconverter for Acquisition Down to 10 MHz	426
Figure 150 ZedBoard Block Diagram for SDR for Plasma Diagnostics	427
Figure 151 : Block Diagram of ARM Processors Through AXI to AD9361 SDR Design Using Zedboard	429
Figure 152 : Analog Devices Companion Downconverter FREQCVT1 for FMCOMMS3 / AD9361 Module.....	430
Figure 153 : Data Flow Rrchitecture of SDR Head with Remote Analysis	431
Figure 154 : TX-RX Loopback 0.999859MHz CW Tone with 12bit I and Q Raw Samples	433
Figure 155: Constellation Plot of I & Q for CW of 0.999859 MHz (not accurate circle)	434
Figure 156 : Some of the Configuration Parameters Available for FMCOMMS3 / AD9361 TX/RX over SPI	435

List of Tables

Table 1 Harmonic / Allocated Frequency Bands.....	45
Table 2 Beehive 100C Frequency Compensation Chart.....	82
Table 3 Mapping from Received Signal Level to Plasma Current and Magnetic Field	87
Table 4 Correction Factor Table for Power (Beehive Electronics Inc., 2019).....	87
Table 5 Experimental Cross Table	186
Table 6 KNN Linear Means Accuracy for Pressure and Gas Flow	203
Table 7 Principal Components and Contributions for Harmonic Measurement Optimization.....	214
Table 8 Experimental Regime Cross Table for Laplace Transform Generation .	225
Table 9 Frequency Cutoff for Differing Modes of the WR430 Rectangular Waveguide.....	287

This page intentionally blank

Abstract

Title: Multivariable Process Diagnostics for Plasma Systems

Author: David Coates

Plasma processing has played a crucial and pivotal role in a number of industries, but probably one of the more significant has been the semiconductor sector. With plasma etching and surface treatment being key steps in wafer manufacturing, modest improvements in methods, methodologies, processes or control at the low level have very significant outcomes in the overall production of semiconductors and their varied uses. This work takes a multi-pronged approach from design and analysis of recently developed radio emission spectra capture to system level data acquisition over multi-sensors (which includes novel sensor approaches) for insertion into a hierarchical time series database structure, built upon a novel message-based network bus subsystem. Early work focused on the development of an optical viewport to obtain a planar view of the plasma. This was followed by the development of the non-invasive E and B-field radio probe technique for plasma monitoring RES (Radio Emission Spectroscopy), leading to the development of two branches of the sensor: the time-domain (TRES) and the frequency-domain (FRES). The two techniques can be applied in parallel to the commonly acquired data, but the processing techniques are somewhat different. The investigation of arcing within the chamber, which is very much a transient event, lends itself to TRES processing. TRES processing is by the application of novel near field B- and E-field analysis to detect these time-variant events where the frequency variation is small. The continuous operation of the plasma can be observed by applying digital signal processing (DSP) techniques to the FRES data, where the main measurement is frequency variant and the time resolution is large. Our approach is to use non-invasive measurement techniques from a range of sensors both in real time and in analytical modes. The analytical mode can be instantiated as an analysis technique, field-deployed, and as such can trigger a direct response to the process. The non-invasive sensors can be used as standalone to monitor operating processes and with simple thresholding can produce notifications of significant events.

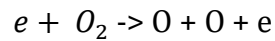
The work on the development of the novel FRES and the TRES sensor systems led to the investigation of the underlying scientific principles for the emission of such waves when driven by a single frequency. This FRES and TRES sensor subsystem has also been a commercial success with the technology being taken up by major semiconductor manufacturers for the determination of plasma conditions while processing wafers.

The combination of novel non-invasive sensors, the real time alarm notification, the analysis over the broad-spectrum data and the instantiation of analytic blocks permits a very powerful and cost-effective tool in the control, maintainability and accounting of the production of semiconductor devices in a modern plant.

This page intentionally blank

Chapter 1 Research Motivation

During the 1960's plasma processing seems to have been first applied to the semiconductor industry via *plasma ashing* (Chapman, 1980). This technique was developed for the removal of the photoresist materials which were placed on the wafer in a previous production photolithography step. The wafers are overlaid with a resist mask using photolithography steps whereby the resist ink is evenly coated onto the wafer and then developed by exposing it to UV light (350 – 450 nm). Modern photolithography uses much lower wavelengths, e.g. deep UV (DUV) 193 nm and extreme UV (EUV) approaching 13.5 nm (V.M. Borisov, 2014)). The areas exposed are set and the remainder can be removed by dissolving in aqueous solution leaving a pattern on the wafer. In the main, the composition of the photo resist materials is organic with hydrogen and carbon being the main constituent parts. There are parts for the differing functions of the resist, such as holding solvent, adhesion promoters, photoactive components and solids. The photoactive components are not soluble in the aqueous base developer once exposed to the activating radiation. In this instance these can be removed using oxygen-based plasmas. In an oxygen plasma discharge, the solid carbon is converted to gaseous carbon monoxide. The gases with the other volatile products of the reaction are then pumped away by the vacuum system. Some of the exhaust includes water vapour. The energy induced by the plasma creates active species and the active oxygen can combine with the resist whilst it will not do so with atmospheric O_2 . The process of electron impact dissociation within the plasma creates this atomic oxygen (Martinu, 2010).



Also, within the plasma discharge there will be an ionization process and these ions in conjunction with the electrons will allow the flow of current and thus enable the continuation of the plasma discharge. With an RF excited plasma this allows the current to flow in a capacitively coupled plasma (CCP), continuing the energy flow to the system. Chapman (Chapman, 1980) describes the glow discharge as an integral part of several processes taking place, which can be grouped together because the type of discharge is relatively common: *“the degree of ionization is typically 10^{-4} , and the current densities are of the order of 1 mA/cm^2 , so that essentially we still have a neutral ground state gas that can be described by gas laws’.* The prime feature of these discharges is that of ionization, with perhaps as many as 10^{18} electron-ion pairs being produced per second.” Plasma ashing is only one of a few techniques widely used in the semiconductor industry.

My introduction to this field had initially been through photolithography, its subsystems and the use of industrial control mechanisms to mitigate against variability in production processes. While the semiconductor industry is very advanced technically and scientifically, it can be observed on occasion that the manufacturing side of the industry falls behind other areas within this otherwise high-tech industry. It is quite a cyclic business with very high volumes and high turnover of designs. This leads to a ‘sweating’ of assets to maximize return on investment. As the latest semiconductor technologies head to 5 nm feature size and the theoretical limit of 3 nm, there is still very much a requirement for larger featured devices such as TTL components, etc. This has led to a multi-level process industry, with some moving towards the ever-finer components and others manufacturing large feature devices (Anton Shilov, 2018, August 27). Some of the

equipment and processes still used in manufacturing are relying on 30-year-old tools and components. The Nikon Body 7 photolithography tools use Digital Equipment Corporation PDP-11 processors, which were built in the late 1980s, for control and monitoring and these are still in use today (Eptek, 2017). The machines for semiconductor processing in the main use Equipment and Materials International (SEMI) Equipment Communications Standard (SECS) / Generic Model for Communications and Control of Manufacturing Equipment (GEM) for equipment to host data communication, but this is mainly for the Manufacturing Execution System (MES). This is used for 'start/stop' control and equipment processing monitoring (Semiconductor Equipment and Materials International, 2011). The standard is very good for monitoring at an individual tool level, but not entirely suitable for parametric sensor monitoring at a low-level for the individual processing of each wafer. The procession of each wafer in and out of a machine can be monitored, but there is very little in the way of process related output at each stage. It is up to the semiconductor plant to install measurement equipment between, or on the various stages to monitor the quality of the processed wafers. The monitoring and analyses of non-invasive sensor data allows the overall control and modelling of the processes impacting the output of the system as a whole. The wafers are processed through multiple stations each performing different actions and these individual steps are then monitored, sifted and analyzed to determine if there are any deviations within the system. The aim is to reduce system variability, improve yields, reduce waste, remove rejects at the earliest possible stage and overall to improve production and reduce costs. A wafer entering the system may have an initial valuation of ~€3K and achieve a full value many times that over a ~10-day period of processing (Eptek, 2017). Any

time saved in removing a non-compliant or reject wafer is a significant cost saving. This reduces wasted wafer processing steps and also frees up the wafer entry point slot for new wafers. Some plants operate a multi-product portfolio in batch mode. Wafers are processed for inventory and single mode continuous monitoring is not as effective (Shi Jian Zhao, 2004, 43). It is now a requirement to reload model parameters for the monitoring to ensure that the recipes have been processed according to process norms for that batch of particular wafers. So not only are there requirements for a multi-variable and derived metrology for process monitoring, but different models must be loaded for the monitoring of the different recipes involved in the batch processing. As an example, the Analog Devices Plant in Limerick Ireland has over 80 different new products added to its production line every year (Doyle, 2017 Mar 29).

We look at capacitively coupled plasma processes and touch upon variations and examine ways to apply novel non-invasive sensors to detect and monitor actions within these processes. The non-invasive sensor methods are also applicable for electron cyclotron resonance (ECR) and inductively coupled plasma systems.

1.1. Multi Variable, Scalable Sensing Systems

Given that the machines for semiconductor processing are multi-generational, one overall engineering requirement for retro-fitted sensing is the non-invasiveness of any solution or at the very least, the minimization of any impact on the current operating procedures. In the case of the Hitachi 712 Electron Cyclotron (ECR) Plasma Etcher, we attached an optically isolated serial cable to a microwave autotuner that matches the source impedance and the load impedance of the plasma and monitored the transition of this impedance matching with the

operation of the recipe (see Appendix L). In any one plant, there may be hundreds of these machines operating at any one time and it is a requirement to accurately and effectively connect the collected data from all these machines acquired in real time over a network, correctly timestamped back to a distributed database for storage and analyses. A complete account of this will be given in Chapter 8. We also want to complete some local computation to produce summary reports of local events which may be propagated at the earliest time into the event-reporting system to permit a timely response to any deviation in the system as a whole.

“The accuracy of a sensor is the maximum difference that will exist between the actual value (which must be measured by a primary or good secondary standard) and the indicated value at the output of the sensor” (National Instruments, 2013, Sep 23). With the drive to produce ever more accurate processes, the boundaries of sensor technology are being pushed. If it is possible to measure more accurately then it is possible to make a more precise product. Thus, in a competitive environment such as semiconductor manufacturing where more precision is always an advantage, then sensor measurements are always ‘of an order’ of the device under measurement, i.e. they measure to such an extent as to be only modestly more accurate than the process under examination with the constraint being the sensor itself. It follows that relying on single sources of measurement can lead to relatively gross inaccuracies. Where we have a multitude or wealth of data sources, these each improve on the classification of events. There are a number of characteristics of multi-sensors that can improve accuracy:

1. Carefully selecting features to minimize error caused by interpreting raw sensor data.

2. Multiple estimates from a single sensor to reduce variance and improve estimation precision.
3. Some classes of faults/events are uncorrelated across multiple sensors, so by combining readings from different sensors (virtual metrology) it is possible to improve classification accuracy.

Thus, the environment is a scalable distributed system with 'processing at the edge'. There is a requirement for processing at the edge to implement local 'analyses' (derived from analytics on the raw collected tagged data) that is pushed from the central repository. Congruently there is an upward flow of rich processed data along with summaries to be stored in a *sharded* (a type of distributed database partitioning that separates the high-volume flows into manageable parts called *shards*) distributed database. We describe the building of a scalable network topology messaging system combined with novel non-invasive sensors for the semiconductor industry and show how processing at the edge can be used to perform analysis at the edge. This analysis at the edge is as a result of being able to perform analytics on the raw centralized data and derive analysis algorithms to detect anomalies. These algorithms can then be 'pushed' to the edge where they can operate at the 'leaf' level. This architecture can be seen in Fig 1 shown below.

Data Acquisition from a Sensor Perspective

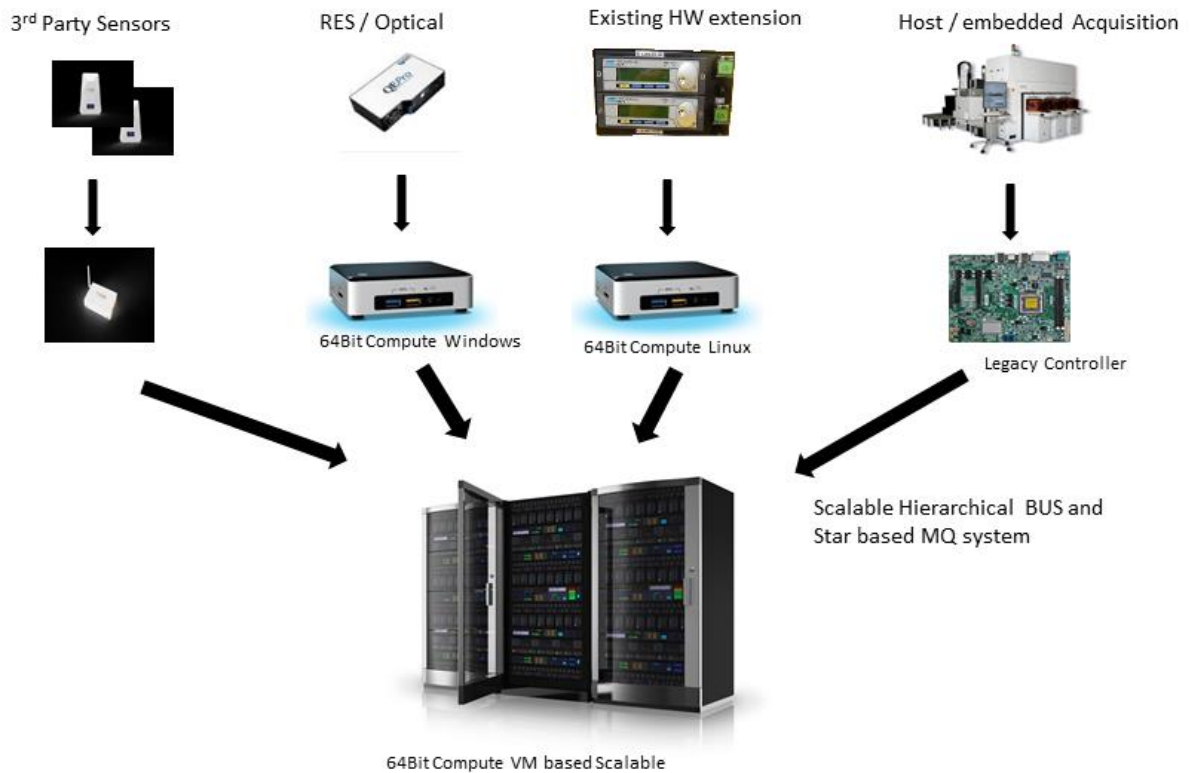


Figure 1 Hierarchical Sensor and Analyses-at-Edge Topology

Figure 1 shows the general sensor flow towards the *sharded* database over the distributed Message Queueing Databus (described further). Analytics are performed across the many streams of data including the derived metrology and analyses algorithms are developed. These are then sent back down the tree to the leaf-nodes (sensor-collectors/data concentrators) to be performed at the edge. This permits distributed processing of the control algorithms located most closely with the recorded data. With the scale of data being present on these individual machines (in the case of Optical Emission Spectrum (OES) acquisition over 4,000 datapoints every second and even more for Frequency Radio Emission

Spectroscopy (FRES) (Kelly & McNally, 2017).), it is not efficient to send these over a network to be processed centrally.

1.2. Research Topics

With plasma processing being prevalent in the semiconductor industry and there being a specific need for non-invasive sensor techniques to solve real-world problems, the focus was on the design of non-invasive sensors and the application of practical acquisition techniques. This permitted the acquisition of high-volume real time data and centralizing of this for storage and analysis. This facilitates the detection of faults in components of the process system and also the output of said process. The individual components are laid out below.

1.2.1. Frequency-Resolved and Time-Resolved Sensors on Oxford Instruments PlasmaLab 100 Capacitively Coupled Plasma Tool.

The analysis of the Frequency-Resolved Radio Emission Spectroscopy (FRES) (Patrick McNally, 2018) data acquired on an Oxford Instruments PlasmaLab 100 Capacitively Coupled Plasma (CCP) tool has delivered high volume raw data that describes a number of properties of the plasma.

The FRES system comprises a non-invasive B-field (or E-field) antenna located adjacent to the plasma (optimally flush with the plasma-chamber wall) and collects the radio frequency emissions from the plasma. The system collects the harmonics of the plasma drive frequency and records them. Multiple harmonics (in the following experiments up to ~100 harmonics of 13.56 MHz drive frequency, necessitating the collection of frequencies from 10 MHz of up to 1.3 GHz). These harmonics were recorded in time and tagged with the operating parameters of the plasma. As part of the project, bespoke software was written

and developed to build the FRES system. This has become the basis for the commercialization of the technology.

Further analysis techniques are then developed based on the data acquired from the FRES system. Also developed, is a Time Resolved Radio Emission Spectroscopy (TRES) which captures the data in real-time and makes real-time analytical approaches to determine events and operating points within the operating plasma. These techniques will be fully developed in chapters 4 and 5.

One of the topics of this report is the examination of this sensor, its operation and the data that can be derived to determine the operation of the plasma. The monitoring of the E- and B-fields adjacent to the viewport of a radio frequency driven (13.56 MHz) CCP in the Oxford Instruments PlasmaLab 100 tool gives a coupling of the inductance within the plasma to the axially aligned antenna. This sensor approach is then extended to the time domain (Time-Resolved Radio Emission Spectroscopy) by examining the use of a Square Law decoder, the Hilbert Transform and the Wavelet Transform for the detection of arcing events or other perturbations. There is also an examination of software defined radios (SDR), with and without zero intermediate frequency (IF) to provide the level of resolution for TRES to be performed. Utilizing the Xilinx Zynq FPGA board with the AD9361 and downconverter, a basic special-purpose SDR was designed for the FRES and TRES (Appendix Q) sensor subsystem.

The TRES analysis system was then extended to permit the observation of events / perturbations within the operating plasma. In chapter 4, novel techniques were designed to transform the captured data into a data space that more accurately describes the transient nature of the operating plasma. This permits the observation of arcing events with the CCP and can be used in an industrial

environment to monitor the correct operation of the plasma tools. Different detector models are built and tested against the recorded arcing data from the operating plasma to show the operational parameters of the detectors. Code for these detectors is included in the Appendices. These are real-time tools than can be implemented on high-speed computer or FPGA based tool-side systems.

We then examine the operation of the plasma as a simulated electrical circuit. This is fully developed in chapter 5. This permits the building of a model that can be readily simulated and compared with the received acquisition from the FRES or TRES systems. The model permits us to perform some element of closed-loop control over the operation of the plasma.

In chapter 6, novel modelling techniques are utilized to correctly determine the operating parameters of a plasma system. The K Nearest Neighbour (KNN) and Principle Component Analysis (PCA) show that with a high degree of accuracy, the real-time operating parameters of the plasma can be determined with a non-invasive sensor subsystem. We introduce a novel way of determining the optimal harmonic set for any particular operating-mode data set within a plasma, so as to optimize resources for the collection, storage and analysis of TRES and FRES data for an industrial plasma.

In chapter 7 we introduce a novel method of building a transfer function of a plasma system. This is combination of its RF drive subsystem to the measured outputs of its harmonics. Using the Laplace Transform method, we drive a system model of the operating plasma, which is a calculable linear equation. This equation may be solved and describes a plane (in the real and imaginary space) which shows the operation of the plasma in a particular operating mode. This calculable

transfer-function may be used as a model to complete a closed-loop control of an industrial plasma system.

In chapter 8 we describe, build and demonstrate a communications system to support the multi-sensor high speed acquisition subsystem that permits the scalable tool-side acquisition back to a sharded data-base and the real-time analyses and control of industrial semiconductor tools. The basic version of this communication system is built in the commercial version of the FRES sensor subsystem.

1.2.2. Event Classification. Processing at the Edge

Given the rate of data production from these spectral sensors, it is a requirement to classify at very high speeds. By producing summary data which is then combined with non-parametric values, this may be classified at high speed using a KNN analysis system. The KNN algorithm may be accelerated using the Intel FPGA SDK for OpenCL (Liu, 2018 Mar 23). This lends the KNN algorithm very well to high speed data classification using hardware accelerators connected to sensors. This is a novel use of high-speed KNN classification for plasma event notification. To the best of our knowledge, this is the first time that an Artificial Intelligence (AI), KNN technique has been applied to radio emission spectroscopy of a plasma system. The KNN classifies according to the *k closest* training values in a hyperspace. Similar algorithmic systems are the Radial Basis Function network (RBF) and the Support Vector Machine (SVM) (Jamimamul Bakas, 2016). The operation of RBF relies on the summing of radial based functions to approximate a given function. In determining the coefficients, the classification can be made. The SVM operates by minimizing an error function to describe classes by

generating a hyperplane to differentiate the classes. The implementation (CPU or acceleration hardware) costs of the latter two are quite high, relative to the former. The thesis will examine the former (i.e. KNN) for performance in the classification task for perturbations in the plasma. The initial targeting of the edge-processing node to an Intel Quark processor that included a Pattern Matching Engine (PME) permitted the building of a hardware supported KNN solution which made it is possible to classify and characterize operations of the plasma out on the leaf. The structure of the hardware PME was emulated in software (see Appendix I). By developing a software based KNN 128X128 PME, it is possible to characterize the throughput of ‘on the edge analysis’ using analytics performed on the centrally stored data. The hardware PME can perform searches at approximately 25,000 samples per second with the main processor clocked at 4MHz. The software emulation, with no optimization could only achieve 12,000 samples per second on an Intel Core i7-3770K processor @3.5GHz using Ubuntu 16.04 and gcc V5.4. The emulation permits the testing of KNN search trees at the edge for distributed analyses, in a multimode industrial acquisition system. The KNN system can be combined with the TRES analytical approaches that we develop. We show the data flow and how the vector output of the real-time Discrete Wavelet Transform of the TRES acquisition is fed to KNN to classify the operation of the plasma system and any perturbations that may occur.

1.2.3. Scalable Data Acquisition in an Industrial Production Environment

By building a scalable message-queuing (MQ) broker-publish-subscribe system, this forms the basis for a network of multi-nodal sensors that may be deployed on existing network infrastructure but utilizing high performance edge nodes

combined with cloud computing at the back end. We will present a working industry deployed MQ based protocol system for scalable acquisition across multi-node sensors in an industrial environment. The bespoke software developed for the FRES system supports this MQ based subsystem as well as a local message token system for non-distributed operation. Bespoke software for an OES sensor from Ocean Optics was also developed as part of this project for data correlation to the FRES system of the plasma parameters. This non-invasive OES acquisition sub-system also supports the MQ based communication subsystem (see Appendix L).

1.3. Summary

For modern volume semiconductor manufacturing, precise control of the resources, the equipment and closed-loop feedback of the underlying processes is of prime importance. We examine these areas, in particular for the industrial plasma systems and develop new and technologies and novel techniques to solve some of these problems. We develop new and unique models for plasma process control and event detection using advanced data transform methods and artificial intelligence components. These are then combined tool-side for multi-sensor acquisitions systems, that permit tool-side closed-loop control and also the forwarding of the high-speed real-time data over the scalable messaging system for later data-analyses and data-mining.

While in an industrial environment, there is ultimately a need for processing at the edge and a means to do this will be explored and facilitated in chapter 8. However, the principal focus of the work is the development of novel non-invasive sensing

techniques, based on the nascent Radio Emission Spectroscopy concept, in order to test their efficacy as a means of understanding and controlling industrial semiconductor plasma processing systems. The chief test vehicle for this will be implemented on an Oxford Instruments 13.56 MHz Capacitively Coupled Plasma Etch Tool. This will be implemented in a stepwise fashion as follows: In Chapter 2 we will develop some basic physical and electromagnetic models for the generation of near and far field radio frequency fields in CCPs. The use of electric field and magnetic field antennae, combined with appropriate software defined radio systems, for the acquisition of Frequency-Resolved Radio Emission Spectra, will be described in Chapter 3. The extension of this technology to rapid (for the Rigol digital oscilloscope 2 ns and for the Agilent 80000 unit 1.6 ps) Time-Resolved RES analysis will be described in Chapter 4 and a set of Transform Methods will be used for the detection and analysis of commonly occurring arcing events within such CCPs. While the capture of such data is interesting *per se*, it truly only becomes useful when the data is understood in the context of controlling the process plasma and/or understanding what the recorded data means in terms of the plasma condition and its potential impact on the wafer(s) being processed. In order to achieve this, there then follows a number of chapters aimed at developing circuit simulation models relevant to the CCPs used in this thesis, especially the modification of these models to represent the captured RES data (Chapter 5, to be followed by a Chapter 6 describing the use of Statistical Process Data Analysis techniques (KNN, RBF, and Principal Component Analysis in particular) to explore whether RES can be used as a non-invasive tool to monitor process parameter excursions in an operating CCP. In chapter 7, the use of the Laplace Transform Modelling provides a method to describe the operation

of the plasma in a calculable equation which may be utilized to close the control loop around a functioning plasma processing system. Finally, in Chapter 8, the data flows from the various tool-side sensors can be brought together for Distributed Scalable Multivariate acquisition.

This page intentionally blank

Chapter 2 Near and Far Radio Frequency Fields of Capacitively Coupled Plasma

2.1. Introduction

As seen earlier plasmas are widely used in industrial processes in a wide variety of guises from contained Plasma Enhanced Chemical Vapour Deposition (PECVD) to open plasma-arc electric welding. Being the fourth state of matter, as the temperature increases, the gases can convert to plasmas. In this state, the plasma is composed of charged atoms/ions, which are mixed with free electrons. This separation permits the plasma to conduct electricity and can thus be an active element in the creation of, and response to, electric and magnetic fields. This feature adds to the complexity of the understanding of the operations of plasmas. In the capacitively-coupled plasma we examine the B near-field radio spectra as the plasma undergoes controlled changes due to the physical operations in the chamber. We also examined the E near-field, but using the same non-invasive observation port to the chamber, it was observed that this signal was significantly attenuated and contained substantial noise. There is a complex changing phase relationship between the E and B near-field. In the FRES experiments, it was possible to only acquire the B near field. (In chapter 3 examining the B-field probes and in chapter 4, examining the time-domain acquisition, we show and analyze the comparative signals. For the bespoke FRES and TRES system we describe the requirement to acquire multi-channel data). The E-field is restricted in penetrating into the bulk and is confined to the thin sheath region, but the magnetic field can extend further (Paul A Miller, 2006). The non-invasive monitored current-flow through the plasma adjacent to the port provides very

good diagnostic information as to the operation of the plasma. The monitoring of optical emissions is well established and is used in industrial machines for process and end-point control (Mackus, Heil, Langereis, Knoops, & van de Sanden, 2010) (Kim, Min, Kang, Cho, & Moon, 2005). The use of B-dot probes to measure directly within the plasma is well developed (Reilly, Miley, & Lewis, 24-02-2009). The plasma itself is the most positive thing in the chamber and the electrostatic shielding of the current probe becomes very important. The probe being immersed in the plasma is not immune from the plasma processing and is therefore also disruptive. The Langmuir probe is such a device using the electrostatic probe method. This comprises a charge-collecting surface inserted into the plasma and has been extended from a simple wire to various geometries. The operation of the sensor is to place the wire into the plasma and bias it with respect of a known potential in the plasma (nominally the conducting surface of the plasma chamber). This device is used to locally sample the charged particle flux by applying an independent bias (Pascal Chabert, 2011). The probe current for an object inserted into the plasma is of the form:

$$I = eA \left[-\frac{n_0 \bar{v}_e}{4} \exp\left(\frac{e(\phi - \phi_p)}{kT_e}\right) + n_s u_B \right] \quad \phi < \phi_p$$

where

A is the area of the surface protruding into the plasma

n_0 is the density at the centre of the sheath

$$\left(\frac{n_s}{n_0} = \exp\left(-\frac{1}{2}\right) \approx 0.6 \right) \text{ (Pascal Chabert, 2011)}$$

$\bar{v}_e$ is the mean electron speed

ϕ is the potential of the probe surface inserted into the plasma

ϕ_p is the potential of the plasma

n_s is the ion density at the sheath edge

u_B is the Bohm speed or ion acoustic speed.

This is for a planar surface, area A . The probe surface potential is at ϕ and this is below the potential of the plasma which is at ϕ_p . With this information, the electron and ion fluxes can be deduced as $\Gamma_{wall} = n_s u_B$ when the sheath is free of ionization, the ion flux leaving the plasma is equal to the ion flux arriving at the wall. The surface retards the Maxwellian distribution of electrons of temperature T_e and only electrons with an energy greater than $e(\phi - \phi_p)$ can reach the probe (Pascal Chabert, 2011). There are a number of assumptions that permit this equation to function as a diagnostic tool, mostly to do with secondary processes being negligible. It is possible to construct a symmetrical double probe and as such, the voltage applied to the probes does not need to be referenced to ground. The probes may be swept differentially from a large positive to large negative potential and the current changes symmetrically between ion-saturation current being collected by one surface through zero to ion-saturation current by the other. This is balanced by the net electron current at the opposing probe. Of note is that these are still in essence, invasive probes as to some extent they are part of the plasma process in the chamber. Thus, there is significant reluctance on the part of management to use such probes in light of the many engineering challenges of inserting active probes into a plasma chamber on a production line. The wafer-processing tools are provided 'as-is' and modification within the manufacturing warranty is a significant concern.

The particular plasma discharges that we are examining for the RF spectral system are driven by high frequency signal generators. They generally have a fixed fundamental frequency (13.56 MHz) and may also have additional bias RF frequencies. The basic mechanical arrangement is shown in Figure 2 below where the high-frequency radio source interacts with the gases in the low-pressure chamber.

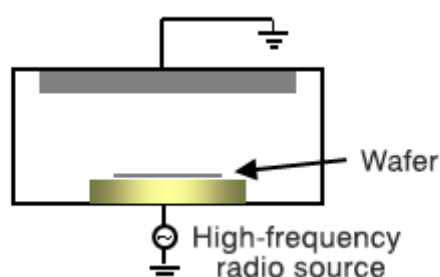


Figure 2 Capacitively Coupled Plasma (CCP)

The basic model schematic for this is shown below but we want to examine this further (Cao, 2016). For high pressures, the electron mean-free path is quite small and the non-linear interaction between the sheath and the bulk is strongly damped (Sebastian Wilczek, 2016). For the range of industrial plasmas that we are interested in for industrial monitoring for semiconductor processing, the low pressures within the asymmetric discharges give pronounced high-frequency oscillations of the RF current. The use of B-field non-invasive monitoring provides accurate measurement of the status of the operating plasma. The asymmetric electrodes of the plasma chamber results in most of the RF power being coupled into the larger sheath at the drive electrode, providing the rich Fourier spectrum of current oscillations. The original circuit representing the plasma was a series RLC by Verboncoeur *et al* (J.P.Verboncoeur, 1993 february)

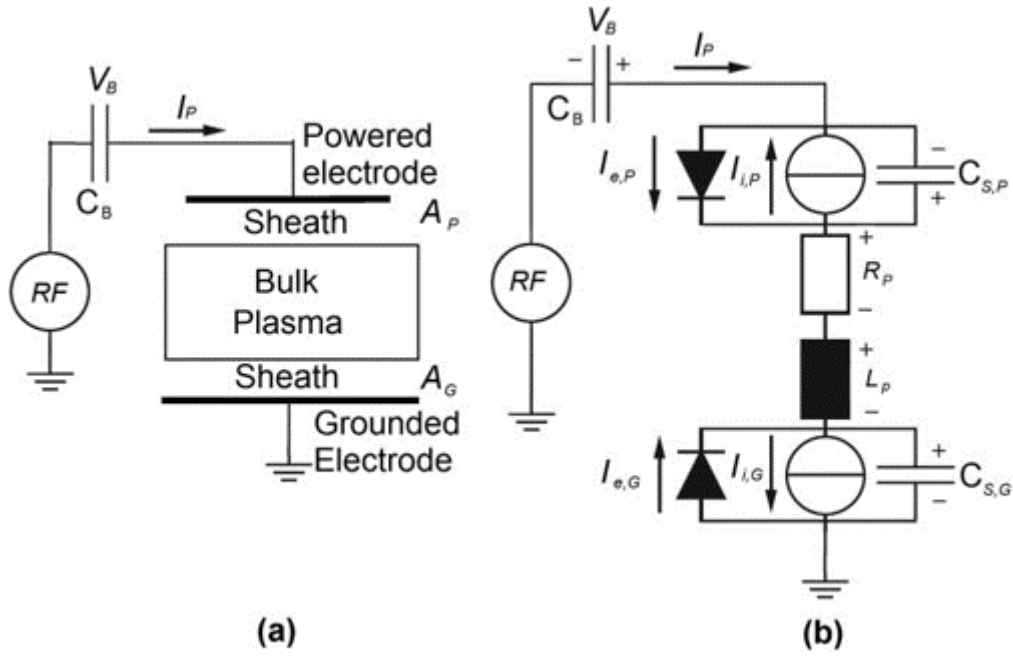


Figure 3 Equivalent Circuit of Capacitively Coupled Plasma

The following components are defined:

- A_G is the area of the grounded electrode.
- A_P is the area of the powered electrode.
- V_B is the bias voltage of the plasma
- C_B is the DC blocking capacitor for the RF power source
- $C_{S,P}$ is the capacitance of the powered sheath
- I_P is the forward current of the RF power supply to the plasma
- $I_{e,P}$ is the electron current within the powered sheath
- $I_{i,P}$ is the ion current within the powered sheath
- R_p is the ohmic resistance of the bulk plasma
- L_p is the inductance of the bulk plasma
- $C_{S,G}$ is the capacitance of the ground sheath

- $I_{e,G}$ is the electron current within the ground sheath
- $I_{i,G}$ is the ion current within the ground sheath.

This equivalent circuit can be used with the non-invasive B-field measurements to quantify physical parameters as they are operating within the plasma. With measurement, the temporal dynamic of the current and voltage at the driven electrode can be measured. This shows ample evidence for the non-linear nature of the system. A sinusoidal applied voltage as with the 13.56 MHz from the power supply unit (PSU) on the Oxford Instruments PlasmaLab 100 tool through the matching unit and on to the plasma leads to strong high-frequency oscillations in the RF current density. In Chapman (Chapman, 1980), we see the description of basic equivalent circuits for RF discharges. A lot of work has been completed by Koenig and Maissel (H. R. Koenig, 1970 Mar) and by many others with the basic circuit as shown above. The sheath is primarily capacitive due to the positive ion space charge in the sheath and the compensating negative surface charge on the adjacent surface. The diode allows for the asymmetry of the ion and electron currents. The bulk is modelled as a resistance and an inductance. The resistance is to allow for power dissipation due to collisions and the inductance is the linear impedance to the applied voltage (Lieberman & Lichtenberg, 2005).

2.2. CCP Machines Under Examination

Initially there were two main machines under examination for FRES. The Oxford 100 tool is used for testing within the DCU campus. Analog Devices Ireland plant very kindly made a Novellus Concept 2 available for test purposes. After

installation at the Analog plant and configuration to obtain Optical and FRES data, commercial agreements were signed with another company for the development of the FRES sensor subsystem and the equipment had to be removed from the Analog plant. Thus, for the purposes of this thesis, we will concentrate on the results of the Oxford Instruments PlasmaLab 100 CCP tool.

2.2.1. Oxford 100 CCP Brief Description

The Oxford Instruments PlasmaLab 100 CCP etch chamber can process SiO_2 and SiN deposition with a maximum wafer size of 8", and has a load-locked chamber for wafer handling convenience. The RF generator is capable of 500W output with an in-built autotuner and a substrate table that can support temperatures up to $400^{\circ}C$. In the Figure 4 below, the chamber and the driven electrode can be seen. Also shown is the B-field antenna and the KF40 port on the tool that is radial to the bulk-plasma location between the surrounding ground electrode and the RF driven one.

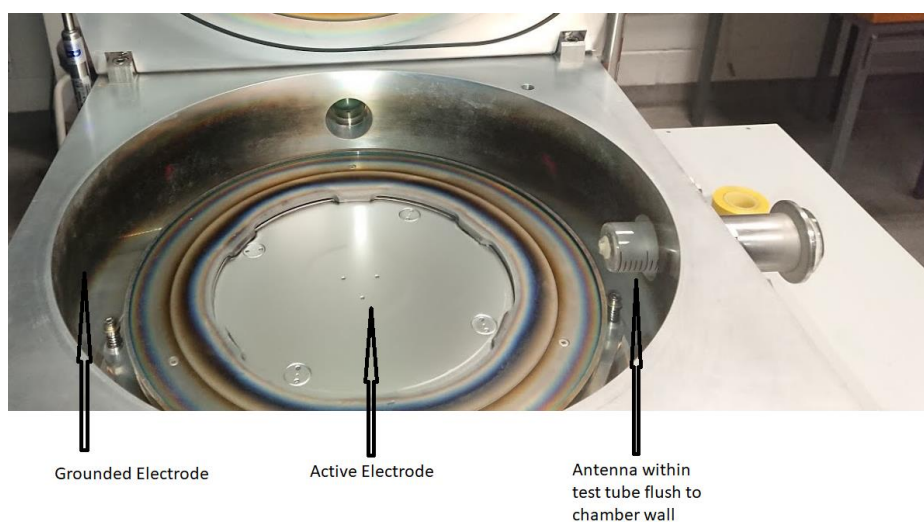


Figure 4 Oxford Instruments PlasmaLab 100 CCP Plasma Chamber with FRES Antenna Location

The Oxford Instruments PlasmaLab 100 CCP was the primary tool for the development of the KNN, the Laplace and Wavelet Transform modelling for use in the non-invasive plasma monitoring for industrial control.

2.2.1. Novellus Concept 2 Brief Description

The Novellus Concept 2 was the initial on-site test system for preliminary data-acquisition using the FRES system. With the commercial agreements in place for the development of the FRES system, the results of this non-core work have been moved to the Appendices (see appendix P).

2.3. RF Operation and the Sheath

There is nothing special about the 13.56 MHz RF power-supply frequency that many CCP systems use. It is based upon a slot in the international telecommunications spectrum where one may radiate a certain amount of energy without interfering with communications. This not such a good solution as determined by the analysis. The plasma discharge has many non-linear effects and generates many harmonics of the fundamental frequency. These frequencies fall within bands that are much more stringent with respect to permitted transmission and emission energies. Initially the predominant interest was in the first few harmonics emitted by the RF plasma when powered by a fundamental drive frequency. Much of the published literature deals with only the first 7 (Sebastian Wilczek, 2016) or 12 harmonics (Frederik Schmidt T. M., 2018) of the fundamental frequency. As can be seen from the RF spectral results, to be discussed later, the interest in the higher harmonics is significantly increased as a

significant amount of information describing events and also the nuanced operation of the plasma is contained within these high-order harmonics. With a patent-pending design of antenna and suitable mounting location, the acquisition in the FRES system is of the first hundred plus (100+) harmonics. These higher harmonics tend to be quite significantly attenuated. We will see with the later analysis how these higher harmonics contribute greatly to the analysis, modelling and detection systems designed. Whilst the second harmonic is of the order of -20 dBm, the 9th harmonic is of the order of -65 dBm. For example, the ninth harmonic lies in the range of the AM based aircraft communications systems and while the signal is attenuated, the applied fundamental RF power is quite significant.

Harmonic (odd/even)	Fundamental (13.56 MHz) MHz	Band ITU (US)
2	27.12	13.41 to 13.57 Fixed/Mobile
3	40.68	40.0 to 42.0 Fixed/Mobile
4	54.24	54.0 to 72.0 Broadcasting TV
5	67.80	54.0 to 72.0 Broadcasting TV
6	81.36	76.0 to 88.0 Broadcasting TV
7	94.92	88.0 to 108.0 Broadcasting FM
8	108.48	108.0 to 117.975 Aero Radio Navigation
9	122.04	121.9375 to 122.0875 Aeronautical Mobile

Table 1 Harmonic / Allocated Frequency Bands

With the FRES, the high frequency RF currents within the plasma are recorded and analyzed. We will see the development of the FRES system using Software Defined Radio (SDR) subsystems initially and then the utilization of Commercial Off The Shelf (COTS) hardware to acquire the data and permit the development of the analysis and modelling for the plasma monitoring and control. The initial phase was to design custom hardware to provide the acquisition speed required for the

harmonic-content analysis both in the time domain (TRES) and the frequency domain (FRES). The use of COTS acquisition hardware, whilst not optimal for real-time industrial use (too slow) and also of single channel design, permitted the development of the software, analyses, modelling and artificial intelligence (AI) classification systems. The KNN modelling, the event detectors, the Continuous/Discrete Waveform Transform and the Laplace Transform, all rely on the accurate acquisition of these harmonics. We show the development of bespoke software for the FRES system supporting COTS hardware. The various software components are contained in the Appendices. Modelling, transform and AI code is included. To support the development of the FRES system, several utility systems were also developed and these are also described and included in the appendices. To distribute the FRES and TRES systems on an industrial scale, a distributed, scalable, managed communications system was developed. This structured component which is included in the FRES, OES and auto-tuner subsystems is described in Chapter 8.

2.4. Remote Sensing of Low-Pressure Plasma in Radio Near Field

The monitoring of radio-frequency wavelengths from a low-pressure plasma is relatively unexploited. The development of radio emission spectroscopy (RES), (Kelly & McNally, 2017) utilizes the analyses of the harmonics of the emitted E- and B-field adjacent to the plasma bulk. The sensor (antenna) can be mounted external to the chamber for a capacitively powered plasma. The sensor is a near-field E- antenna and/or a B-field magnetic loop type. The magnetic loop type is

approximately 25 mm diameter that intercepts the magnetic flux resulting from the currents in the plasma bulk, as shown below.

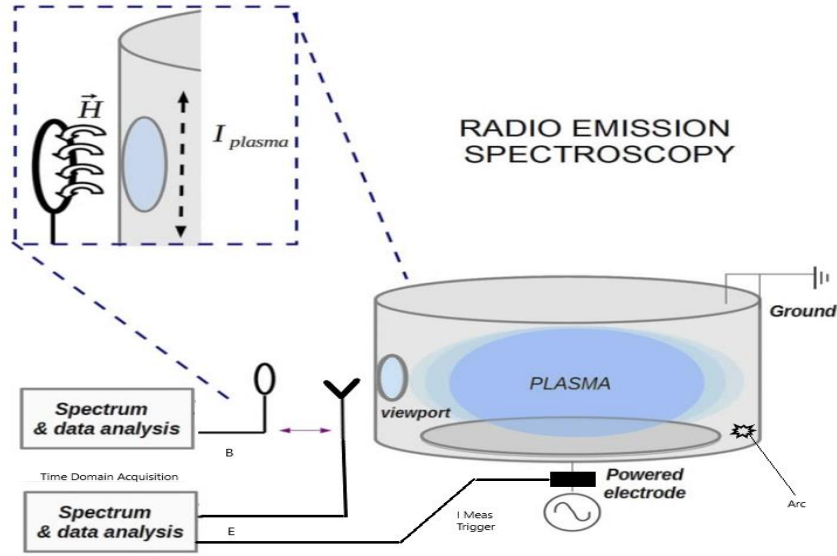


Figure 5 Near Field Magnetic Coupling (Kelly & McNally, 2017) (In their original concept there was no direct or indirect coupling to the powered electrode)

The magnetostatic currents flowing in proximity to the view port are given by the Biot-Savart law

$$B(r) = \frac{\mu_0}{4\pi} \iiint_v \frac{J(dV) \times r}{|r|^3}$$

where dV is the volume element for the current density J , r represents the spatial distance of the antenna and the current volume, and μ_0 is the magnetic constant or permeability of free space ((CGPM), 2019). The antenna forms an inductive element and produces a voltage in relation to the impinging magnetic flux and the number of turns, in this case one. See chapter 3 for further information on the Beehive Antennae (Beehive Electronics Inc., 2019) and also the custom designed plain-coil RF Identification (RFID) tag antennae that were utilized for the experiments. Note that there is a frequency correction for a practical implementation of an antenna. For the Beehive 100A antenna, there is a 5 dB gain

from 10 MHz to 20 MHz where the primary operating frequency of the plasma is 13.56 MHz. Rather than calibrate for unknown antennas, a calibrated unit was utilized for the experiments. The later FRES experiments employed a custom plain-coil RF ID tag antenna and the differential harmonic measurements were utilized for plasma characterization. In the experiments for the ensuing AI modelling, transfer functions were performed on the low-pressure chamber RF data from the Oxford Instruments PlasmaLab 100 tool. This chamber incorporates a turbo pump required to pump down to the lower pressure regimes. The initial antenna measurement experiments were performed in an Ar plasma with various powers operating between 50 watt and 500 watt and with pressures varied from 20 to 50 millitorr. Later experiments for the gas-flow AI classification and Laplace modelling were performed using an O_2 plasma varying the pressure between 20 and 50 millitorr and the gas flow rate from 20 sccm to 30 sccm. Figure 6 below shows the general configuration of the acquisition system.

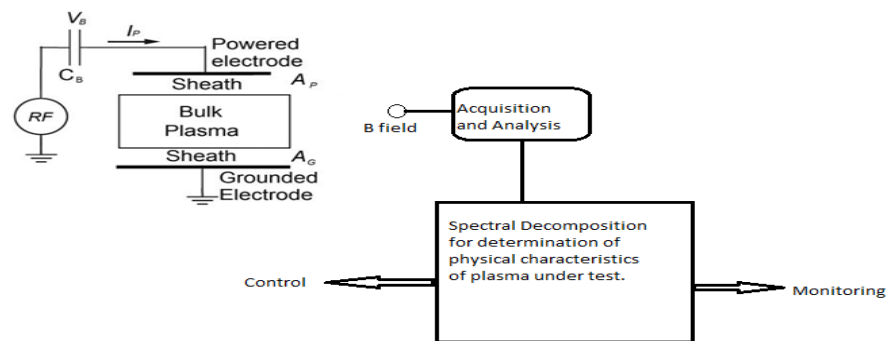


Figure 6 General Overview of FRES Acquisition

Using a SignalHound SA44B Spectrum Analyzer ([Signalhound](#)) and its acquisition software with the B-field antenna permitted the acquisition of the high-frequency

harmonics while the Oxford Instruments PlasmaLab 100 CCP was running a recipe. SignalHound's Spike™ software has a recording facility and with the analyzer in 'sweep' mode, acquires a sweep of frequencies and records these to a BBR™ file. Later developments include a patent-pending antenna design and also a companding-technique for the signals for optimal acquisition from a RF CCP. These are described Chapter 3.

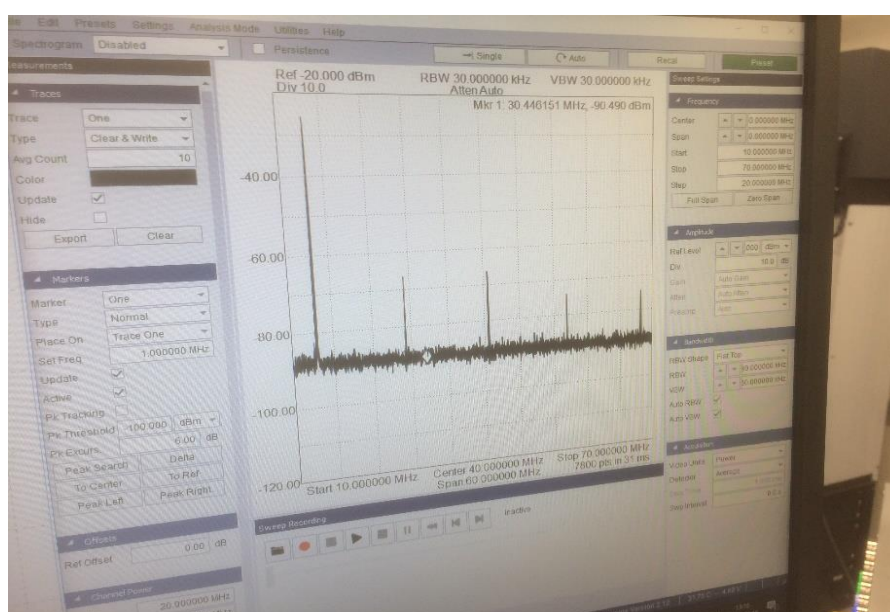


Figure 7 Screen Shot of FRES Acquisition Using SA44B Spike Software (reproduced by special permission)

To reiterate, the low-level signals were acquired using a SignalHound SA44B spectrum analyzer and stored in BBR proprietary files. The low-pressure plasma system was operated with different experimental regimes and the corresponding RF harmonics emitted by the plasma were recorded using the antenna located adjacent to the KF40 port with the SA44B analyzer and software. Bespoke software was written (see appendix B) to convert the recorded files into a format that could be imported into analytical tools such as Matlab®. This recording

system was not very user friendly nor conducive to the task of accurately acquiring the relevant harmonics to a high accuracy required for the analysis. The Oxford Instruments PlasmaLab 100 CCP tool possesses a 13.56 MHz capacitively powered plasma source. We can associate the signal power measured in the antenna with the current flowing within the plasma, though the location of the antenna has an averaging component to the measured B-field (Paul A Miller, 2006). The plasma was run through a number of cycles with different recipes and the SignalHound SA44B was put in Record mode. This produced a number of BBR files, the format of which is native to the SignalHound analyzer. The recorded files were processed using the software utility which was written for both Windows and for Linux to transform the recorded data in BBR files into CSV files for importation to Excel or Matlab. See Appendix B.

The Biot-Savart law above describes a magnetic field generated by a stationary electric current. With the charged particles in motion within the plasma, there is a time-varying electric current. There are three sets of particles in a plasma: ions, electrons and neutrals. These all have various masses and temperatures, though in the plasmas that we are dealing with, the temperatures are not exorbitant, e.g.

$$T_e \approx 2eV, T_i \approx 0.04eV \text{ and } T_n \approx 0.025eV \text{ (Chapman, 1980).}$$

where the suffices “e”, “i” and “n” refer to the temperatures of electrons, ions and neutrals, respectively. (It is interesting to note that energies can be converted to an equivalent temperature. Using the Boltzmann constant, one finds that energy $E = k_B T$, where $k_B = 1.38 \times 10^{-23}$ J/K, and T is expressed in kelvin. Thus $1 \text{ eV} \approx 11,600 \text{ K}$). For our overall neutral plasma, the ion and electron densities are approximately equal, but this number is usually much less than the number of neutrals. On average, electrons have a velocity far in excess of the neutrals or the

ions mainly due to the low mass of the electrons. The rapid motion of electrons with respect to the plasma permits them to move away and in doing so, they leave behind a relatively positive charged plasma (Chapman, 1980). One possible result of a particle striking a surface is for an electron to be ejected. Within the plasma containment and the surfaces of the anode and cathode, there are incident particles. The number of electrons that could be ejected for an incident particle is called the secondary electron coefficient. Notwithstanding the secondary generation that can add additional electrons (V. Baglin, 2000), there is a broad current drift through the bulk of the plasma and a resonant drift.

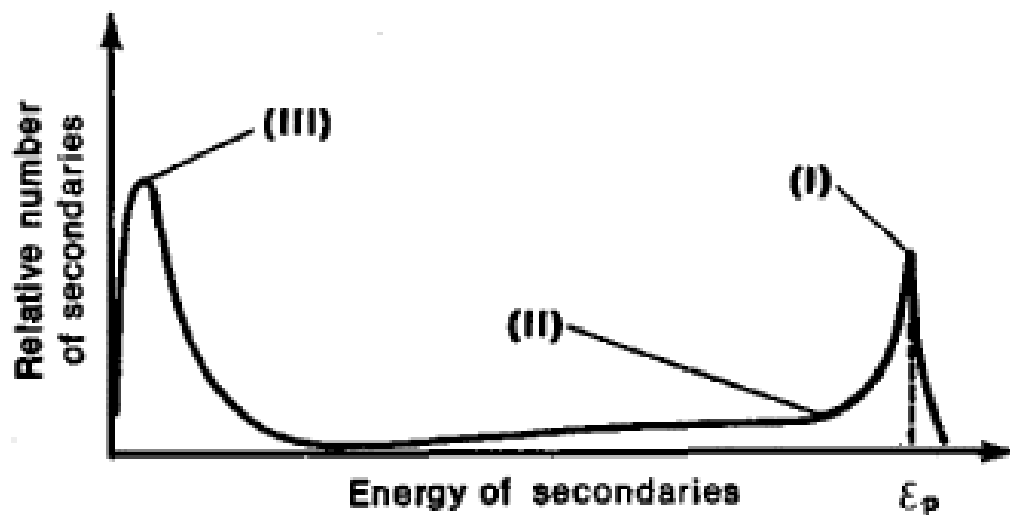


Figure 8 Distribution of Secondary Electrons Emitted by Silver According to Energy of Impinging Electron (Rudberg, 1930) (reproduced from (Chapman, 1980)).

In Figure 8 above, we can see the energy distribution of secondary electrons emitted by silver (Rudberg, 1930). Area (I) shows the elastically reflected primaries. The energy of these is of the order 160eV. Area (II) shows the inelastically reflected primaries and these of the order of 100eV and finally area

(III) shows the true electron secondaries where the energy is of the order of a few eV. The impinging energy ϵ_p is the primary electron energy. What is interesting is that secondary emission is not insignificant and while not applicable to the plasma's emissions that we are examining, silver can provide a yield of up 1.5 times that of the primaries for example.

An approach to the self-bias of an insulating electrode is given by Butler and Kino (H. S. Butler, 1963). We know that a probe at floating potential draws no net current and if a potential is applied to the probe, the current drawn will be given by the probe characteristic. The probe will be struck by the positive ions and electrons, but the flux will be unequal due to the much higher velocity of the electrons and will take on a negative charge with respect to the plasma. This charging will continue until a balance is reached between the negative repulsion and attraction to positive ions. This potential of the probe is the '*floating potential*' which maintains an equal electron and ion flux to and from the probe. With a dc-blocking capacitor on the powered electrode (see Figure 9),

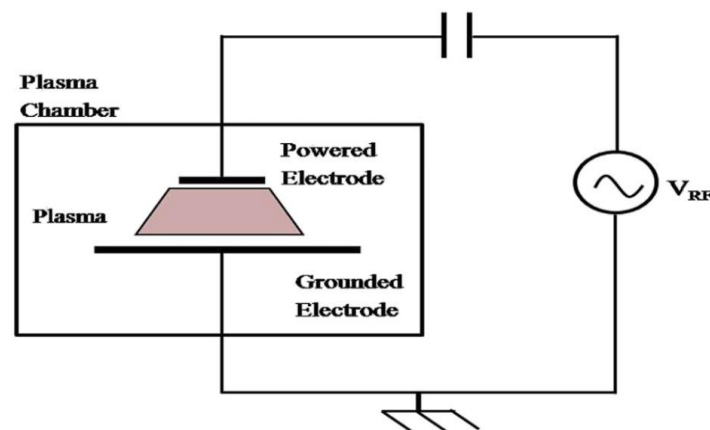


Figure 9 Capacitively Coupled Plasma, dc Blocking

the total net charge flow per RF cycle must sum to zero. The powered electrode will also attain a '*floating potential*'. This is shown in Figure 10 below:

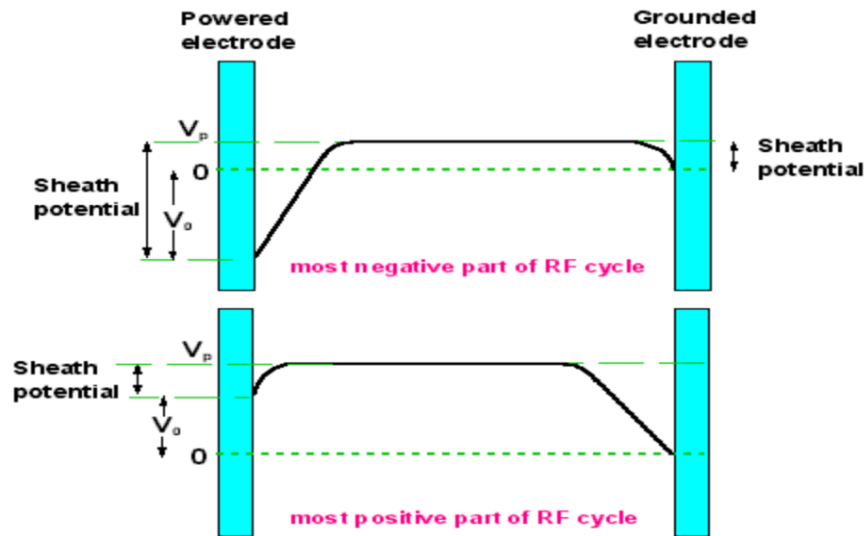


Figure 10 Sheath Potential Asymmetric Electrode at Positive and Negative Parts of RF cycle (May, 1997)

This Figure 10 indicates the relative potential gradients from the plasma to the electrodes at the extreme ends of the RF cycle. With a zero-charge flow over the RF period, there is a very different sheath potential at each electrode due to the time-averaged potential distribution. This is further complicated by the non-symmetrical plates where the driven plate is much smaller than the grounded plate. This enhances the potential across the sheath at the powered electrode. This may be shown diagrammatically as in the figure below (Figure 11):

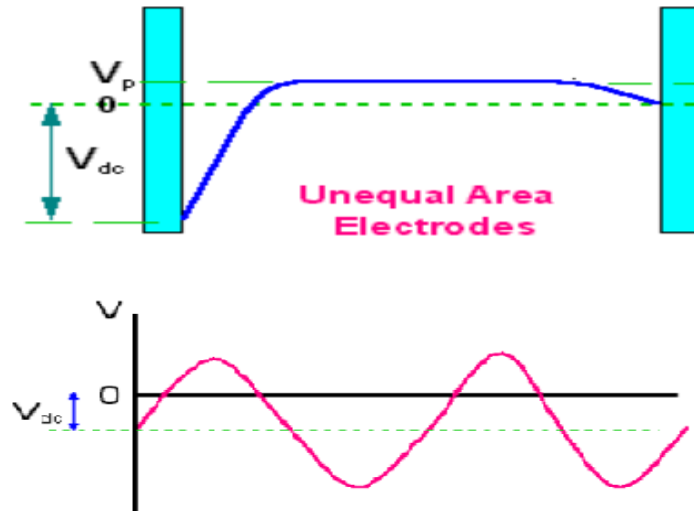


Figure 11 Unequal Electrodes, with 'Floating Potential' and Time-Averaged Potential.

As we have seen in the picture of the Oxford chamber above (Figure 4), the tool has two asymmetric electrodes. The Advanced Energy Hilight 136 RF generator in the Oxford Instruments PlasmaLab 100 CCP tool drives the matching unit and this contains the dc-blocking capacitor which then powers the drive electrode mounted in the bottom of the chamber. The powered electrode (as in the case of the Oxford Instruments PlasmaLab 100), is usually much smaller than the grounded one, charges negatively with respect to the plasma until there is zero charge flow over the RF cycle. This '*floating potential*' (V_{dc}) as shown in Figure 11 is of the order of 450 V for the Oxford Instruments PlasmaLab 100 CCP tool when it was performing the recipe for the experiments.

The voltage across the sheath influences the energy with which the ions strike the substrate (again there will be secondary emission). The density of ions and electrons is sum zero as the Coulomb charge sums to zero. The ion enters the sheath with low energy and is accelerated across it by the sheath voltage. If there are no collisions, it would strike the substrate with an energy-equivalence of the

sheath voltage. It is possible to determine a variation around a voltage perturbation. This can be caused by the rapid movement of electrons and the relatively slow speed of ions, see Figure 12 below. With the potential at $x = 0$ defined as ΔV_0 relative to the plasma, the potential varies with x . We have a measure of ΔV_x which we assume is more negative than V_p (the plasma bulk). An offset of potential of ΔV at location $x=0$ (single dimension) is established which is more negative than the plasma bulk and in this case only energetic electrons can enter this area. A net positive charge will form in front of this charged surface as only the energetic electrons will enter. The ions are too massive to react rapidly to this charge disturbance and it will be only for a very short time period. The ion density to a first approximation in the sheath is n_i as in the undisturbed plasma. If the perturbation of ΔV_0 was maintained for long time this would not be true. We look at where this perturbation is for a very short timescale.

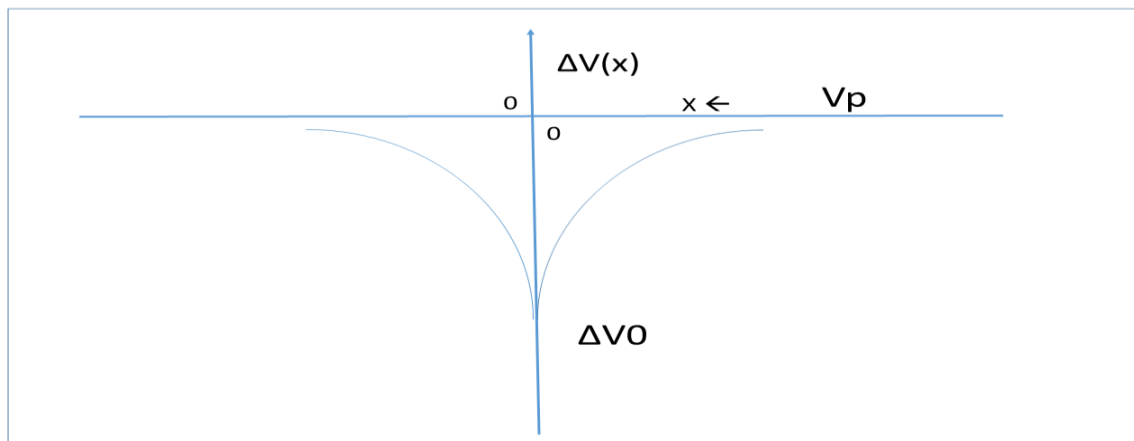


Figure 12 Variation of Potential around a Perturbation

We see that there is a length where there is a volume influence for a floating substrate within a plasma discharge. This length is termed the Debye length. A floating substrate will begin to charge with the high arrival velocity of the

electrons with respect to the ions and at some point, it will begin to repel electrons and begin to attract positive ions. The ion flux is limited by the random arrival of ions from the sheath-plasma interface. Ions entering the sheath do so with very low energy and are accelerated across the sheath. This energy they gain would be equivalent to the kinetic energy with which they would strike a floating substrate. If the numbers of ions and electrons in a plasma are very large, but equal, then it would be seen that the net charge is zero. There is obviously a response time for a perturbation within the plasma. The ions are too massive to respond quickly, but the electrons are much more so. This can be described using a differential equation derived from Poisson's equation for the charge distribution (Chapman, 1980):

$$\frac{d^2V}{dx^2} = - \frac{en_i}{\epsilon_0} \left(1 - \exp - \frac{e\Delta V(x)}{kT_e} \right)$$

where ϵ_0 is permittivity of free space.

The distribution of charge must satisfy Poisson's equation and if the system is in thermodynamic equilibrium, then the charge concentration is described by the Boltzmann Distribution. This gives rise to a non-linear equivalence. This solution to this equation is more apparent when the initial state is $\Delta V(x) \ll kT_e$ (Mitchner, 1973). The solution from this gives rise to a 'distance' approximate solution where

$$\Delta V(x) = \Delta V_0 \exp \left(- \frac{|x|}{\lambda_D} \right)$$

The quantity λ_D , the Debye Length, is a dimension of length and tells us that for a perturbation the plasma attempts to correct the charge imbalance, but that this sphere of influence is within a *distance range*. Thus, for calculations of net interactions, the sphere of influence lies approximately within one or two Debye

lengths. Beyond this, the first order interactions become immaterial. Note that as $|x| \rightarrow 0$, the field remains as that of a single charge ΔV_0 . For distances greater than λ_D , the field dies off exponentially. Charge fluctuations may occur over distances smaller than λ_D . Solutions to this non-linear equivalence may be made with some assumptions, but one solution leads to a characteristic length scale of (Morton Mitchner, 1973):

$$\lambda_D = \left(\frac{kT_e \epsilon_0}{n_e e^2} \right)^{1/2}$$

where:

λ_D is the Debye-Hückel length

k is Boltzmann's constant

n_e is the density of electrons

ϵ_0 is the permittivity of free space

T_e is the temperature of the electrons.

The quantity λ_D has the dimensions of length and is known formally as the Debye-Hückel length but is typically shortened to the "Debye Length". As stated previously, within a value 1 to 2 radii of the Debye length, plasma perturbation interactions are material, but outside this distance, the net interaction is zero. By way of example, for a plasma with density of

$$n_i = n_e = 10^{10} \text{ per cm}^3 \text{ and } kT_e = 2\text{eV}, \text{ the value of } \lambda_D \text{ is } 105 \mu\text{m}.$$

The electrons and ions are, as a whole, in equilibrium, but with a small perturbation from neutrality, there is a large restoring force. With the relative inertia of the ions with respect to the electrons, it will be the electrons that will respond first and this is proportional to the displacement. This is just the set of

conditions for oscillations. These oscillations are determined by the interaction between the ions and electrons and for electrons is related to the Debye Length λ_D (Mitchner, 1973). This relationship can be explained as the movement of an electron over the distance λ_D in the time $1/\omega_e$ which is the minimum frequency for propagation of a wave in the plasma. For practical purposes, the frequency is of the order of hundreds of MHz to ~ 1 GHz and due to the inertia of the ions, the equivalent frequency of ions is of the order of a few MHz (Chapman, 1980).

2.5. Sheath Formation

From the discussion above, we can see that a screening charge cloud forms around a voltage perturbation. If we consider a sphere of 1 to 2 Debye Lengths centred on a point, outside the sphere of influence the net interaction is zero and the plasma is equipotential. This does assume that the Debye Length is much smaller than the overall dimensions of the plasma. The difference in ion and electron thermal energy means that electrons are lost to the electrodes at a faster rate than the ions.

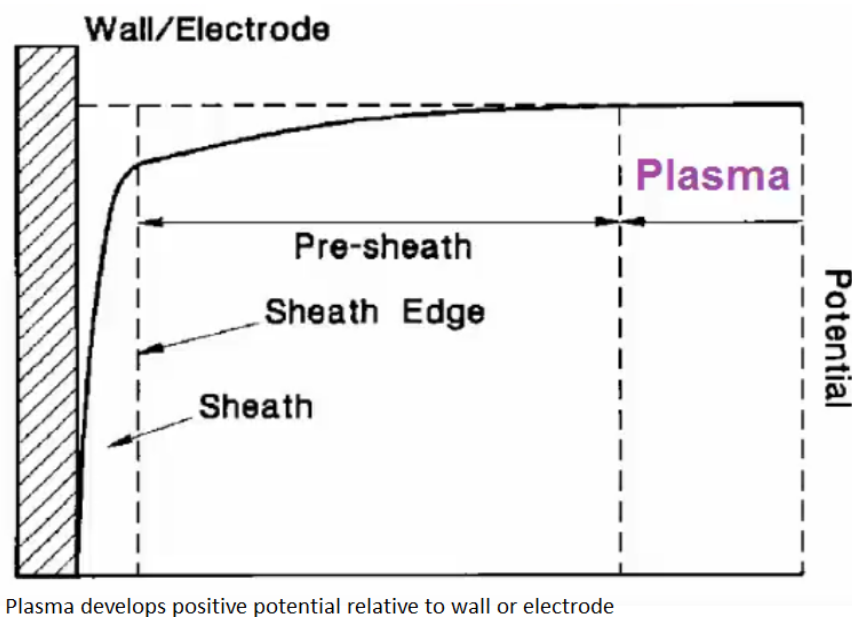


Figure 13 Sheath Formation at Floating Substrate

The electrons are repelled by the potential difference $V_p - V_{dc}$ and therefore the area around the will acquire a net positive charge. This is generally referred to as the '*space charge*' and forms a sheath. The sheath has a density of charges, and this is referred to as the '*space charge density* ρ '. With a net positive charge, there is a reduction of electron density in the sheath and as the number, density and energy of exciting electrons is smaller, the number of relaxing atoms is smaller and therefore the sheath does not 'glow' as much. The sheath does not terminate where the ion and electron densities become equal, instead there is a quasi-neutral zone between the plasma bulk and the sheath proper. This is referred to as the '*pre-sheath*' and is shown in Figure 13 above. This region has a low electric field which increases the velocity of ions before they enter the sheath, crossing the sheath-edge (Bohm & Gross, 1949) (Severn, 2007/01/01).

The simple model is that the resulting electric field falls off in an exponential manner and reduces to the thermal energy of the surrounding electrons and ions. Once at this position, the ions and electrons are free to move away from the charge cloud. This would suggest that the edge of the charge cloud at the Debye length (λ_D) $\Delta V \sim kT_e$. The calculation of the Debye length above assumed $\Delta V \ll kT_e$ so the screening effect has to be carefully applied when considering the sheath formation. It was considered that the sheath consisted of a region where the ion and electron densities became equal. Bohm (Bohm & Gross, 1949) demonstrated a quasi-neutral transition region of low-electric field and the effect of this region is to accelerate the ions entering the sheath at the sheath edge as indicated in the Figure 13 above. This was revisited by Sternberg (Sternberg & Godyak, 2007) and

they suggested that Bohm's results were misunderstood, by concluding that the edge of the sheath coincides with the reference point of the sheath-plasma model. If there is an object that is negatively biased with respect to the plasma bulk, the effect of the Bohm criterion is to describe the increase of the ion flux to overcome that. The ion velocity on entering the sheath must be greater than $\left(\frac{kT_e}{m_i}\right)^{\frac{1}{2}}$ which is determined by the electron temperature (Bohm & Gross, 1949) (Chapman, 1980). This is a quite a strange result, as it indicates that the ion velocity on entering the sheath must be greater than a fixed function of the electron temperature. The physical significance of this is that the ion density must decrease less rapidly than the electron density across the sheath to meet the criteria of acceleration of the ions and repulsion of the electrons. This can be graphed showing the cases where u_0 ion velocity at the sheath-edge is greater than and less than the criterion with a linear monotonically decreasing electron density

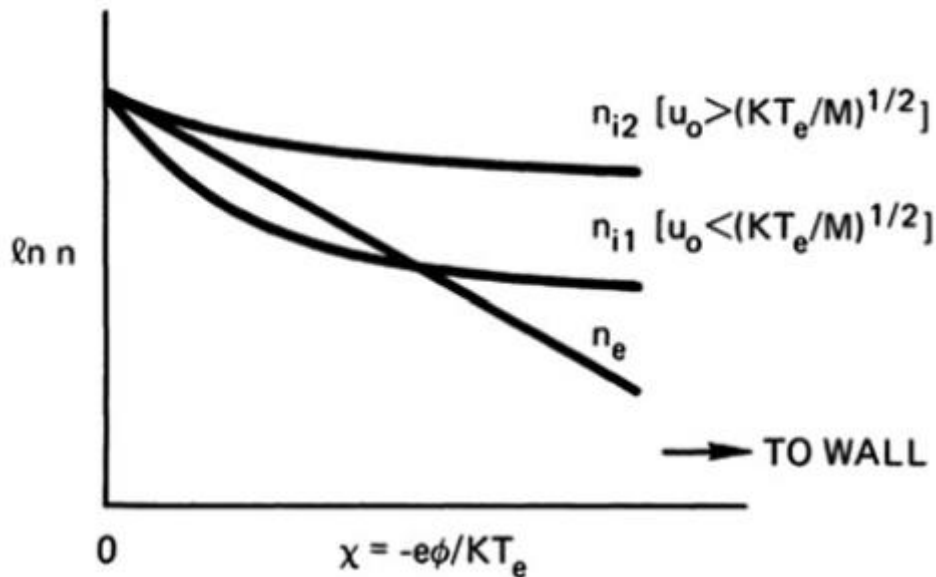


Figure 14 Variation in Ion and Electron Density with Potential $V(x)$ in a Sheath, reproduced from (Chapman, 1980).

The figure (Figure 14) shows the variation of ion and electron density with potential $V(x)$ in a sheath, for two cases: μ_0 greater than and μ_0 less than the critical velocity $\left(\frac{kT_e}{m_i}\right)^{\frac{1}{2}}$ (Chen, 2016).

There must be an electric field across the this presheath region so as to give the ions a directed velocity of μ_0 towards the electrode.

2.6. Harmonic Generation

In Miller (Paul A Miller, 2006), using a B-dot probe inside a CCP plasma chamber, recordings were taken of the B-field across the bulk plasma close to grounded electrode sheath. The harmonic content for the B-field was higher, closer to the centre of the plasma radially than at the edge. In the figure (Figure 15) below, this is shown as normalized harmonic magnitudes in relation to radial position within the plasma.

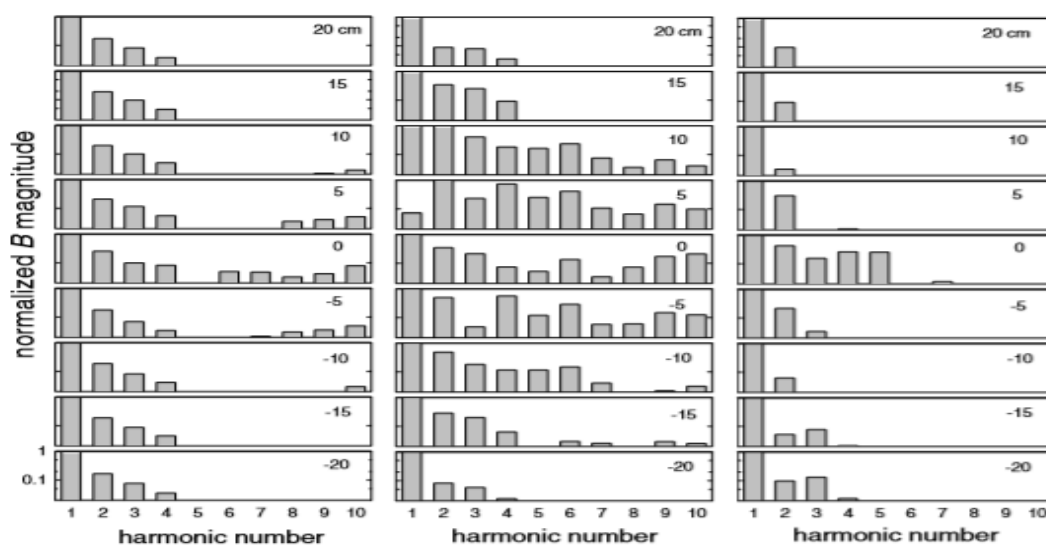


Figure 15 Fourier Transform of B-dot Signals (Paul A Miller, 2006)

The figure (Figure 15) depicts three sets of graphs for excitation RF frequencies of 13.56, 60 and 176 MHz, respectively. Each graph shows the harmonic magnitude on a logarithmic scale with the amplitudes normalized to the largest one.

There is also a complex phase relationship between the harmonics radially across the plasma. Miller shows this in a graph (Figure 16) for the first three harmonics.

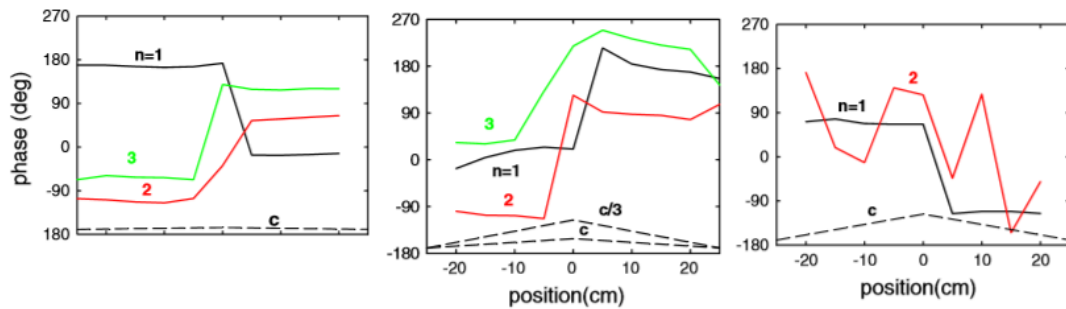


Figure 16 Phase Relationship for the First Three Harmonics across the Plasma (Paul A Miller, 2006)

This indicates a spatial relationship to the production of the harmonics and their phase relationship. We have seen how the wave propagates outwards and the magnetic field is not as constrained by the skin-effect as the electric field. At low discharge currents, the electrons are quite monotonic in their energy spread, as higher discharge currents are applied, and thus there are electrons with both higher and lower energies and the spread is much greater (Langmuir & Tonks, 1929). Tonks and Langmuir subsequently found high frequency oscillations in the plasma. As in the *klystron* faster electrons begin to catch slower electrons, but because of repulsion, tend to bunch-up. The net result is that electrons tend to vibrate about the position that they would have occupied without any modulation. A simplification of the harmonic generation is that of the diode-mixer with a space-charge step-recovery giving rise to the comb-frequency generation. We have seen

the basic dual diode equivalent circuit for the operation of the capacitively coupled plasma and the non-linearity of the transfer function of the diode can be used as a frequency multiplier. Non-linear Transmission Line Multiplication (NLTL) can also function as a comb-generator. With just varactors and inductors, a transmission line in which the propagation velocity is voltage dependent (Spikings, C.R.; Seddon, N.; Ibbotson, R.A.; Dolan, J.E., 2009) can operate as a harmonic frequency generator.

Our primary focus is the acquisition and recording of these high-order harmonics such that they can be analyzed to perform monitoring and control operations over the plasma in an industrial environment. Of note is the variation in harmonic power content based upon the drive RF frequency (Figure 15 above). With multi-frequency chambers, the KNN modelling hyperspace and the TRES discrete wavelet vector generation will be quite dispersed. The analysis for optimal signal acquisition will be of prime importance (see Section 6.6) in these cases for the differing drive frequencies and chambers.

2.7. Summary

The non-linear operating function of the sheath in the CC plasma is a crucial interface and the interactions that occur in the space contribute greatly to the overall dynamic actions of the plasma. Being able to measure parameters of these interactions in real-time using non-invasive methods, is of great importance to industrial plasma processing operators, as this can provide a wealth of information on the operation of the plasma and the processing that it is performing.

This page intentionally blank

Chapter 3 Software Defined Spectral Analysis for B- and E-Field Plasma Analysis

3.1. Introduction

The IC industry is loath to introduce any further equipment that might have an effect any of the processing steps in a production environment. Therefore, any sensor that that can make indirect measurements is to be generally preferred. By having an antenna external to the plasma system and being able to derive process data is of great benefit. Some of the benefits may be listed as:

- Non-invasive, so as not to perturb the plasma itself.
- Can capture real-time data for precise control of plasma parameters.
- Repeatable measurements aids tool matching.
- Can be implemented with COTS initially.
- Rich source of real-time data for FRES and TRES. (e.g. Endpoint detection).
- Can be combined with other sources of real-time plasma data (e.g. OES) for orthogonal data sets to augment analysis of plasma operation.

Therefore, the ability to attach a non-invasive sensor to existing systems that can provide a rich data stream of the plasma operation is of great benefit in an industrial production environment.

3.2. Radio Spectrum Analysis: Non-Invasive Plasma Monitoring.

There has been much discussion in recent years about software defined radios (SDR) (Collins, Getz, Pu, & Wyglinski, 2018). Some of this is due to the immense advances that have been made in semiconductor technology and some are due to

the flexibility of the forms and functionality that they can take. There are some basic characteristics that make them unique with respect to other types of radios. We use the term radio here when we are considering the spectral analysis that are performed on the near-field plasma radiation for the capacitively-coupled plasma under investigation. As the name implies, an SDR is a radio / analyzer that has the ability to be implemented using components of software or logic for the fundamental signal flow. This can be performed using digital signal processors or field-programmable gate arrays (FPGAs).

To take advantage of these digital components, the basic analogue signal must be converted to the digital domain and back to the analogue domain as required. This conversion is performed by high-speed analogue-to-digital (ADC) and digital-to-analogue (DAC) converters. To make best use of this digital processing, the signal is maintained in the digital domain for as much as possible (Rauscher, Christopher, 2014). In this constrained domain, it retains its essential properties. This permits digital techniques to be used to perform functions that would be normally be implemented by analogue circuits. Some of the digital functions cannot be readily implemented in the analogue domain and this permits the testing of functions that cannot otherwise be performed.

Whilst it might seem feasible to digitize the antenna, it is preferable to design an analogue front end (this is the basic circuitry to collect the signal from the device under test) that is more selective and sensitive to a wide range of individual frequencies and then tune the ADC to the bands that it can readily process. In the design for the non-invasive plasma sensor, there are two fundamental frequency ranges of interest. These are constrained because of the machines available. (There are a number of different frequencies and biases on the different plasma

machines available: e.g. 400 kHz, 450 kHz, 3.0 MHz, 13.56 MHz 27.12 MHz 0.915 GHz and 2.45 GHz. This does not represent an inclusive list and then there are the combinations of these frequencies, where there is a plasma generation frequency and ion-energizing frequency. (Reece Roth, 1995)). These are the harmonics of the capacitively-coupled 13.56 MHz powered plasmas. For the plasmas we are examining, we will describe the implementation of spectral analysis of frequencies from 13.56 MHz and the available harmonics produced by the non-linear medium that is the plasma itself. There is a requirement for high selectivity / discrimination of the observed frequencies and their harmonics.

3.3. Design Requirement of Non-Invasive E- and B-Field Analyzer

To remotely monitor the B-field radiated by the plasma, a high-speed integrated RF spectrum analyzer was designed. The basic design is to monitor the B-field non-invasively through the observation window on the device under test (DUT) and use this to determine the operation of the plasma. There are some particular requirements of the system for plasma monitoring and these are discussed further. We are especially interested in frequency domain components for the average processing of the plasma or time domain components for event determination. With the former, it is possible to build a software defined radio (SDR) and process blocks of acquisition in batch mode, and with the latter, build a continuous acquisition and processing system to see high speed events in the plasma.

3.4. Software Defined Radio System

The overall block diagram for the SDR is indicated below. The basic form is to have an antenna system with a suitable analogue wide-band amplifier / impedance matching subsystem followed by conversion to the digital domain. See Figure 17 below.

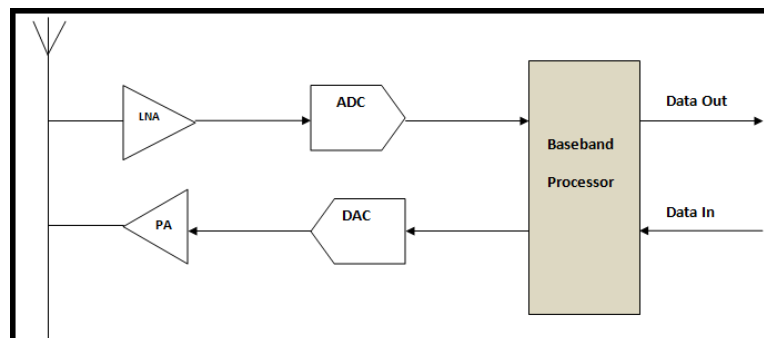


Figure 17 Basic Block Diagram of Digital Radio System

If we can sample directly at Nyquist sampling rates higher than the frequency of interest, this is called a zero intermediate frequency (ZIF) design. Alternatively, a local oscillator (LO) is required with a mixer to move the band of interest to within the range of the acquisition ADC. For the purposes of recording the fundamental, and say, the first five harmonics of the Oxford Instruments 13.56 MHz etcher system, the acquisition is required to acquire from approximately 10 MHz to 70 MHz. The initial design utilized the RTL2832U device (Kelly & McNally, 2017). This is a USB based mixer frontend combined with high speed ADC producing IQ samples on a USB streaming interface. The bandwidth is limited by the USB data rate and the signal accuracy by the ADC data width. The basic block diagram of the USB RTL2832 subsystem is shown below (Figure 18).

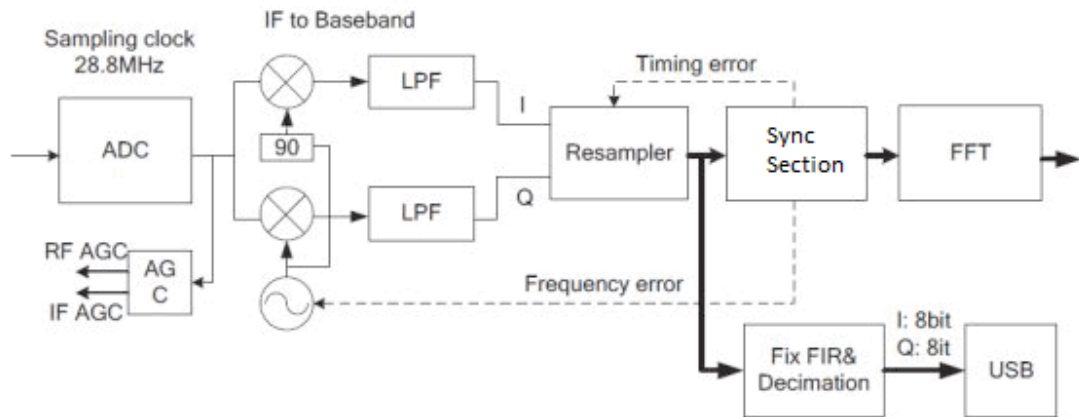


Figure 18 Initial SDR Non-Invasive FRES design

It is very important that the 90° phase shift between the I and Q signals is very accurate. A few degrees in phase difference or even phase jitter can introduce very large errors into the acquisition and introduce harmonic distortion. To this end, by using monolithic acquisition front ends rather than designing from discrete components, can the errors be contained. There are also other factors that can affect the sampled signal quality. The ZIF quadrature sampler is affected by dc offset, phase noise in the LO, quadrature skew in the two arms and gain imbalance in the quadrature filters (Ellingson, 2003).

Work was begun on the design of a bespoke SDR for FRES and TRES. The design is based on the Analog Devices series of high-speed acquisition devices utilizing a LO front end and an FPGA interface for processing the real-time acquisition data. The prototype system based upon a Zedboard Xilinx Zynq and AD9631 is described in the Appendix Q together with the test procedures that were undertaken.

To progress the commercialization and advance the analysis of the FRES and TRES system, it was decided to utilize COTS components and to develop bespoke

software to speed up the development cycle. This permitted the acquisition of experimental data and subsequent development of algorithms for TRES plasma event detection systems and the FRES plasma classifications systems. These are more fully developed in the following chapters.

3.5. Commercial Off-The-Shelf (COTS) Based System Components

For TRES, the data was acquired with a Rigol 500 MHz digital quad-channel oscilloscope and at a later stage with the Agilent DSO 80000 10 GHz digital scope. To examine the operation of the plasma for FRES, the SignalHound SA44B spectrum analyzer was utilized. We will see in a later chapter that this device was augmented with a Tektronix RSA306 spectrum analyzer for higher fidelity for the upper frequencies. With the SignalHound SA44B's own software application programming interface (API), bespoke software was developed for a high-speed (distributed) collection system. This and development utilities were developed for the collection and analysis of data capture from the Oxford Instruments PlasmaLab 100 CCP system. A suite of file-transform tools was developed on Linux and Windows to collect and convert the captured B-field spectral data to more readily processed files. This included development of conversion tools to take BBR and Tektronix TEK files into Matlab® and Excel® for post-processing. The software code, together with brief functional block diagrams can be found in Appendix B.

3.5.1. Brief Description of the SignalHound SA44B Spectrum Analyzer

The SA44B is a USB based SDR receiver which is corrected for temperature and has a calibrated input with a software API that permits bespoke acquisition to be

performed (SignalHound, 2017, Version 3, May 4th 2018). One of the problems to overcome with the Oxford Instruments PlasmaLab 100 CCP system and its 13.56 MHz base frequency is that the harmonics are quite widely spaced in the frequency scale, i.e. 13.56 MHz, 27.12 MHz, 40.68 MHz, 54.24 MHz, 67.80 MHz, ... The internal IQ demodulator captures approximately 2MB of data each second, but this is hardware bandwidth limited to 250 kHz. The I/Q demodulator has dual 24bit ADCs (truncated to 16 bits) (SignalHound, 2017, Version 3, May 4th 2018) which contributes to the wide dynamic range of the RF front end. This is required as the nominal voltages seen from the B-field antenna at the observation port of the Oxford Instruments PlasmaLab 100 CCP are of the order of -80 dBm (being of the order of 20 μ V p-p). The SA44B is not suitable for signal measurements with a signal modulation above 250 kHz and thus would not be suitable for the TRES (see lowpass filter outputs for Oxford Instruments PlasmaLab 100 CCP at 13.56 MHz collapse for arcing in Section 4.1). The modulated frequencies are all well above the base frequency of 13.56 MHz, so the SA44B would be unable to acquire them. The SA44B uses a programmed LO to scan a single 250 kHz bandwidth and then reprograms the LO to cover the next 250 kHz bandwidth within the overall scan range required. In the particular case of the Oxford Instruments PlasmaLab 100 CCP and its 13.56 MHz base frequency, this results in multiple acquisitions 'stitched together' to cover the four or five harmonics of the scan (10 MHz to 70 MHz). The SA44B has been shown to be very useful for the initial measurements for FRES, but note must be taken into account of the limitations of the device. The bespoke software developed for use with the SA44B permitted a selection of centre-frequencies to be defined and the software continuously re-programmed the SA44B to 'hop' along these frequencies. Each acquisition was 250 kHz in

bandwidth centred to cover a particular harmonic. In this way, the software recorded a vector of frequencies, centred around each harmonic of the 13.56 MHz drive frequency. The gain can be set for each harmonic so that the maximum dynamic range is available for each harmonic to detect the smallest change possible. With the Oxford Instruments PlasmaLab 100 CCP, particular attention was paid to the parameters being applied to the device when it was being operated in acquisition mode for the high-speed IOT monitoring function using the SignalHound SA44B API.

3.5.2. Spike Software for the SignalHound SA44B

The Spike software is a software utility supplied by SignalHound to operate the SA44B as a spectrum analyzer. We utilize it to make recordings of spectra of the B-field of plasma in the Oxford Instruments PlasmaLab 100 CCP system. These files are stored as proprietary *.bbr* files. We then post process the files by converting them to timestamped CSV with a descriptive header. The BBR convert utility was developed to convert these proprietary recordings into files that could be analyzed using the standard Matlab and Excel tools. The source code for the tool is included in the Appendix B.

3.5.3. BBR and High-Speed Acquisition

For the Oxford Instruments PlasmaLab 100 CCP, the primary frequency was based around 13.56 MHz (bar frequency corrections for load autotuning) and its first four or five harmonics (13.56 MHz, 27.12 MHz, 40.46 MHz, 54.24 MHz and 67.80 MHz) which in this case is up to 70 MHz. By recording the B-field from 10 MHz to 70 MHz (40 MHz centre with 30 MHz span) over known recipe runs, the *.bbr* files

were created and later converted to CSV for analysis. Later with the development of the bespoke acquisition software, the SA44B was programmed to 'hop' from centre-frequency to centre-frequency and the vector of acquisition was stored directly in CSV files for direct processing.

3.6. B-Field Antenna Subsystem

Non-contact probes are most suitable for a production type environment where there is no requirement to make any changes to the machine under observation. The probes do not measure the voltage or current directly, but rather through the electromagnetic fields that surround a conductor or in this case, the current flowing through the plasma. E-field probes measure the electric field from the plasma as a result of the current flowing through it and H-field probes measure the magnetic field. The H-field and the B-field are closely related, where H is measured in units of amperes per metre and B is measured in teslas or newtons per metre per ampere. A common relationship between the two is given, in vectoral format as $\mathbf{B} = \mu \mathbf{H}$, where the magnetic permeability of the medium is μ . Where the antenna is located outside the chamber, the magnetic permeability is approximated by the *in vacuo* value, i.e. $\mu_0 \approx \mu_{vac}$, and the H-field probe measures the magnetic field and the magnitude of the H-field is proportional to the current flowing axially through the plasma. With a low frequency, the impedance of the plasma is quite high (Lieberman & Lichtenberg, 2005) and falls off with increasing frequency (GE & Zhang, 2014)(see Figure 19 below), thus the relative magnitude of the E- and H-fields can vary.

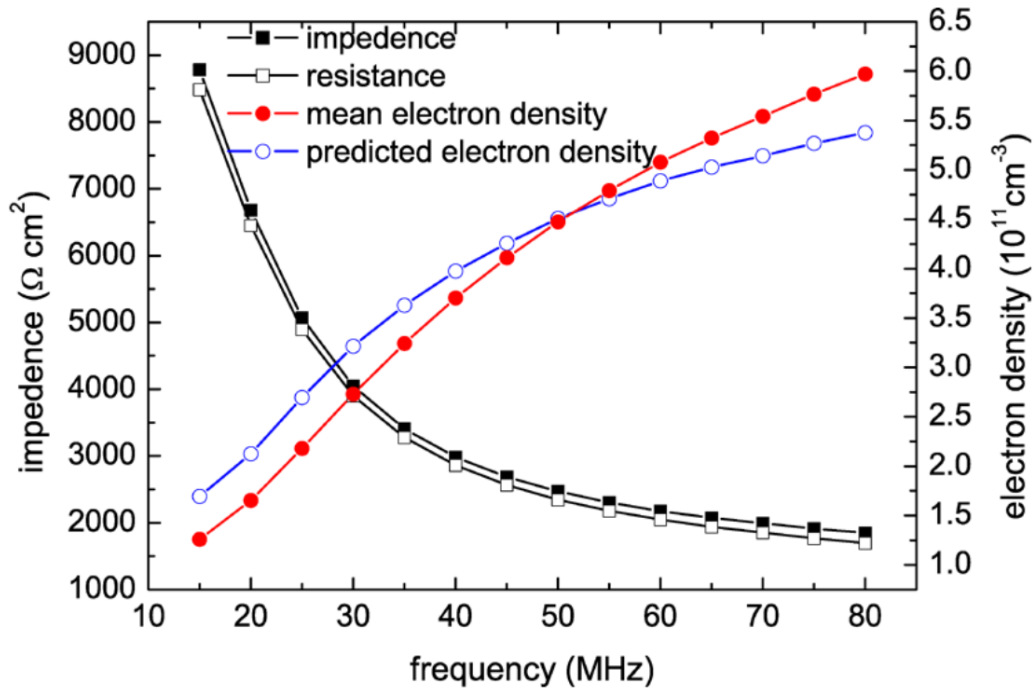


Figure 19 : Predicted Impedance for Atmospheric Plasma with Constant Power Density 50W/cm³ (Ge & Zhang)

In the low impedance operation of the 13.56 MHz plasma shown above in this data from Ge & Zhang, the voltage will be relatively low and the current will be high and the H-field probe will display better sensitivity. The design of the B-field loop is such as to electrostatically shield the E-field and only develop a signal for the applied B-field. The near field is the radius $<< \frac{\lambda}{2\pi}$ of the source and the far field is considered radius $>> \frac{\lambda}{2\pi}$ from the source (Johnson, Ecker, & Hollis, 1973), where λ is the wavelength of the RF radiation. For 13.56 MHz and the first few harmonics, the wavelengths λ are of the order of 22 metres. One of the defining characteristics of the near field is that the nature of the wave depends upon the source characteristics and that it is a high current, low voltage source, then the source is mainly magnetic and has a low impedance (high H). For the far field, the nature of the wave depends more upon the propagation medium and moves to the

impedance of free space of 377Ω ; additionally the electric and magnetic fields are both planar waves and orthogonal to one another (Morgan, 2007). This may be graphically represented by the following diagram (Figure 20):

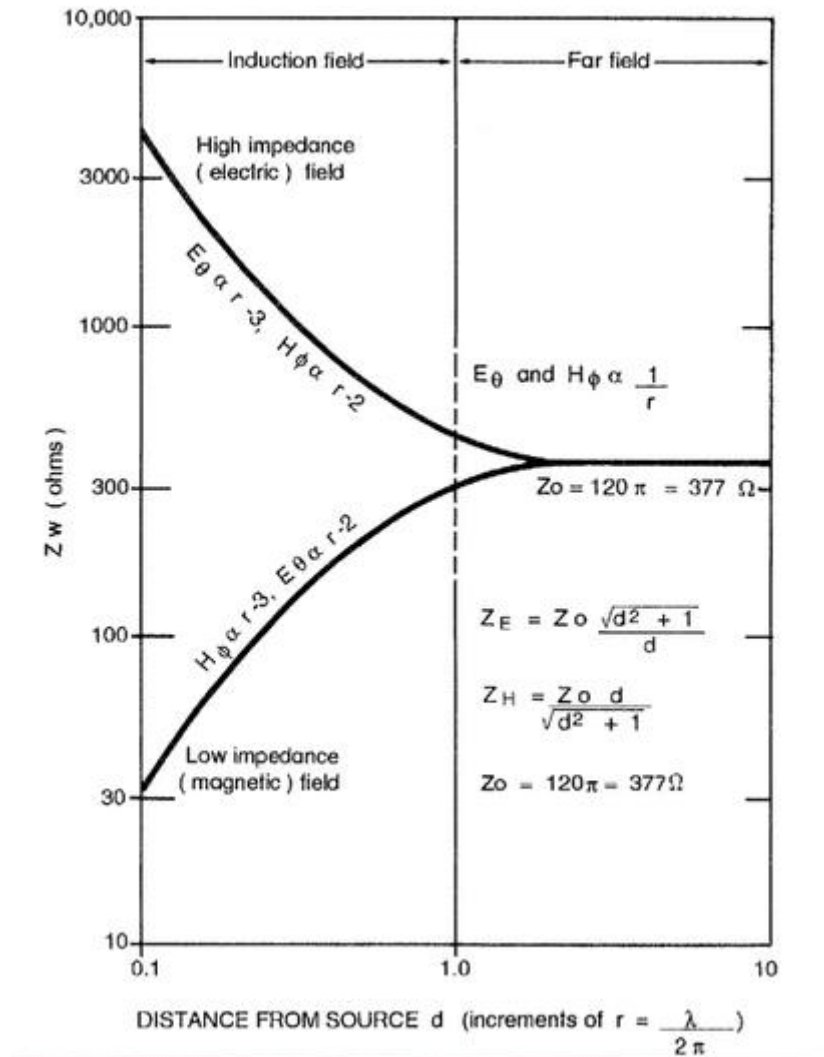


Figure 20 Variation of the Wavefield Impedance with Distance from the Source (Morgan, 2007)

Absorption of radiation in the far field does not affect the load on the transmitter, but in the near field it does. In this case, the source is the operation of the sheath and the bulk current in the plasma, but the absorption by the antenna is negligible in comparison to the energy scales the plasma of the Oxford Instruments

PlasmaLab 100 CCP. This will be demonstrated further in Section 3.8 of this thesis. There is a form of magnetic induction between the antenna and the conduction in the plasma bulk and operation of the sheath, most likely via displacement currents and non-linear effects (as discussed in Section 2.1). Individual far-fields fall off with a

$$\frac{1}{r}$$

dependence, where r is the distance from the source such that the total energy $(|\mathbf{E}| \times |\mathbf{B}|)$ falls off as

$$\frac{1}{r^2}$$

The near field has non-radiative components as a result of radiation scattering off an object. The near field dominates close to the source with the E component decreasing as $\frac{1}{r^2}$ and the B component as $\frac{1}{r^3}$ (Charles Capps, Delphi Automotive Systems, 2001, August 16). The antenna, being adjacent to the observation window on the Oxford Instruments PlasmaLab 100 CCP tool, is thus relatively insensitive to a second chamber when it is operating. (There is some pick up of the RF from the adjacent chamber as described in Section 3.10 below. The unbalanced cables from the antenna subsystem may pick up the adjacent chamber RF signals. This is addressed by a novel patent-pending design). This is very important from an industrial application point of view. Inter-chamber modulation would render the system inoperable, as the signal levels for the received high-order harmonics are at a very low level. Other adjacent chambers operating on the same frequency could interfere with these measured harmonics. It is very important in an industrial environment to be able to discriminate the signals from those of the

adjacent operational machines. The operating distance was observed to be approximately 10 cm (Patrick McNally, 2018) from the port of the Oxford Instruments PlasmaLab 100 CCP system (at an operating power of $\leq 500\text{W}$). Later measurements were made with the antenna mounted flush to the inside wall of the chamber and the reception of even higher-order harmonics was possible. With the B-field antenna in combination with the SignalHound SA44B, the first 103 harmonics were recorded. This is examined in more detail in the analysis in Chapter 6 and Chapter 7.

3.7. Design of Loop Antenna

The particular loop antennae that we are using with the Oxford Instruments PlasmaLab 100 CCP were fabricated on a printed circuit board (PCB) with stitched impedance matched 50Ω track. The basic circuit is shown below with the PCB layout (Figure 21). The stack layup of the PCB permits the shielding of the inner core.

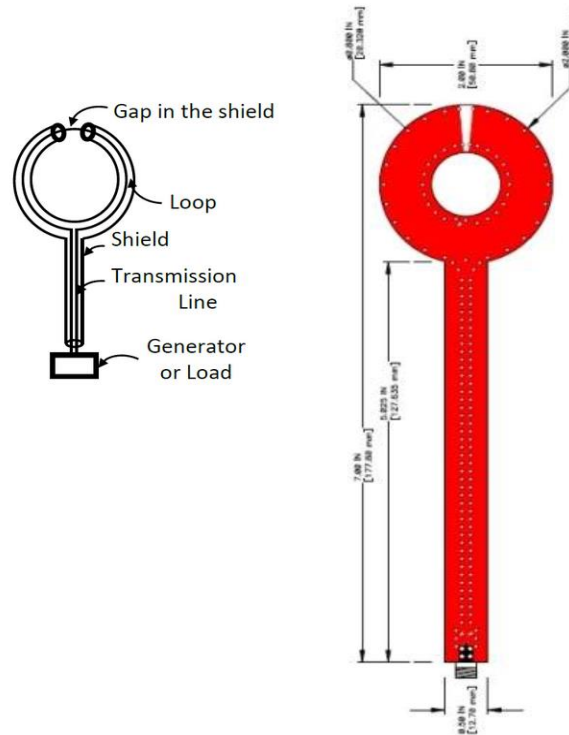


Figure 21 PCB B-Field Antenna Construction

The basic design is as per Ediss, Philips Semiconductors (R Ediss, 2018) where the loop is made with a track on a multilayer PCB for the enclosed signal and the ground loop is through-hole stitched around it. The circuit is tuned for the desired frequency of operation. The principle of operation is that a current is induced in the skin of the outer conductor which appears across the sheath gap in the loop. The outside track width is much greater than the skin depth. There is no induced voltage on the inner conductor. The 50Ω microstrip shields the inner conductor from the E-field.

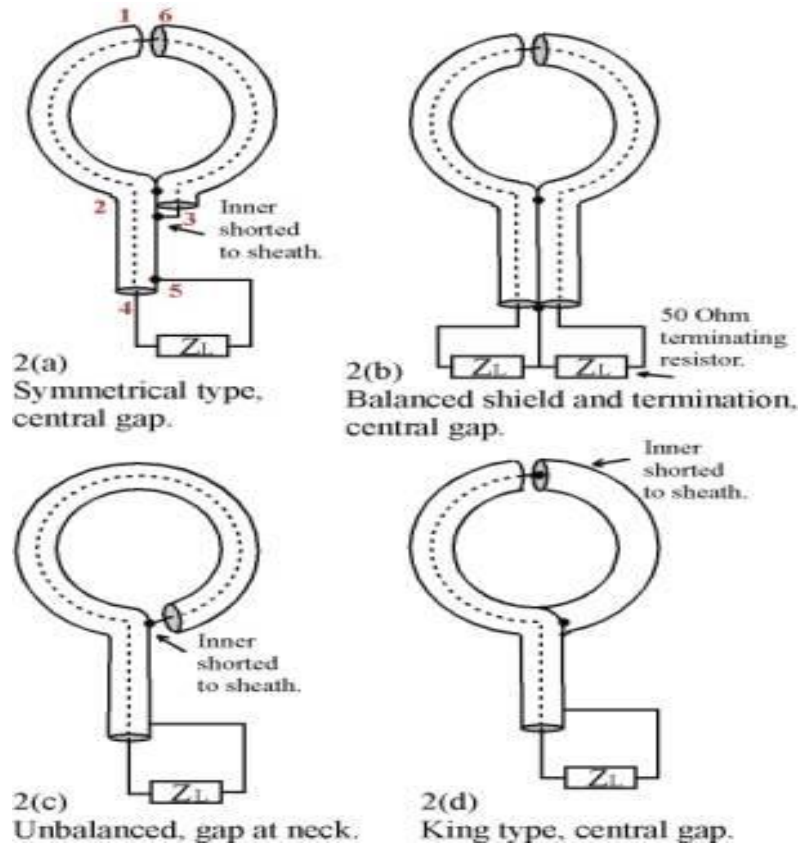


Figure 22 Generic Types of Magnetic Field Probe (R Ediss, 2018)

For the initial measurements using the SA44B, the Beehive range of B-field and E-field probes were used. These are examined in more detail in the section below. For the FRES classification in Chapter 6 and the system modelling in Chapter 7, a bespoke 13.56 MHz loop coil antenna was assembled and utilized. The use of this antenna permitted the installation of the antenna through the chamber port, flush with the chamber wall and the relative harmonic measurements rather than their absolute value was the desired measurement.

3.7.1. Mutual Inductance

When two coils (in this case the loop B-field antenna and the natural inductance of the plasma) are arranged with their axes in the same line, a current sent through the plasma will induce a magnetic field which cuts through the loop of the antenna.

Consequently, an electromotive force (emf) will be induced in the outer sheath which is carried to the inner sheath which itself is electrostatically shielded. Since this current appears in the second coil (antenna), it is a 'mutual' effect and results from the mutual inductance between the two coils. The relative mutual inductance is quite small as not all the magnetic lines from the plasma cut through the antenna. The maximum coupling is reduced by a ratio of actual mutual inductance. As negligible power is drawn from the antenna, and the mutual inductance is low, the power generated in the antenna which must be opposed by the primary inductance is such that the power in the plasma will only be the 'magnetizing power' of the plasma inductance.

The Beehive antennae may be purchased with an optional calibration certificate for an additional charge. This calibration certificate lists the frequencies and output power of the antenna at 1 microtesla field strength. The measurement is made using a Hewlett Packard HP 8714C vector network analyzer, a HP 85052D mechanical calibration kit, a HP 809C slotted line and a Beehive Electronics ET103 Helmholtz coil to generate the magnetic field. An example calibration certificate is shown in Figure 23 below.



8555 Lawrence Lane
Sebastopol CA 95472
(707) 824-9206
info@beehive-electronics.com

Calibration Certificate

Model number:	100C		
Serial number:	3004		
Calibration date:	12/1/2014	time:	3:33 PM
Operator Number:	1		

Calibration Equipment Used

Manufacturer	Model	Description	Tracking number	Cal Due Date
Hewlett-Packard	8714C	Vector network analyzer	107	1/1/2015
Hewlett-Packard	85052D	3.5 mm cal kit	121	1/1/2015
Hewlett-Packard	809C	Slotted line	ET102	6/15/2015
Beehive Electronics	ET103	Helmholtz coil	ET103	6/15/2015

Frequency (MHz)	Output power at 1 microtesla field strength (dBm)
0.100	-54.80
0.125	-52.86
0.150	-51.28
0.200	-48.78
0.300	-45.26
1.000	-34.80
3.000	-25.65
6.780	-18.78
10.000	-15.62
20.000	-10.24
27.120	-7.98
30.000	-7.26
100.000	-1.35
300.000	-2.39

Probe condition:

As found:

Cosmetic condition: pass
Impedance: pass

As left:

Cosmetic condition: pass
Impedance: pass

Calibration method:

Absolute accuracy calibration performed at 1 MHz using vector network analyzer and Helmholtz coil.
Frequency response calibration performed using vector network analyzer and slotted transmission line to generate the test magnetic field.

Environment:

The laboratory is maintained in a temperature controlled environment with ambient conditions from 18 to 28C and relative humidity less than 90%.

Traceability:

The test equipment listed above is traceable to SI units through the National Institute for Standards and Technology or other recognized national metrology institute.

Figure 23 Beehive 100C Loop Antenna Calibration Certificate

This gives a characteristic curve to match a known magnetic field, the frequency and the output of the antenna into a 50Ω load. Using the datasheet and the

characteristic calibration certificate for the BeeHive 100C, A and B loop antennae the following chart was drawn up. The particular area of interest is the first 15 harmonics of the Oxford Instruments PlasmaLab 100 CCP tool, which would fall into the *flat* range of the 100C (25.4 mm) loop antenna, which is 10 MHz to 300 MHz.

The frequencies are selected to best match the characteristics of the loop. For the Oxford Instruments PlasmaLab 100 CCP tool with 13.56 MHz the Beehive 100C antenna was chosen. This has a relatively flat response over the range of 20 MHz to 300 MHz.

Frequency in MHz	100C (25.4mm) O/P at 1 μ tesla (dBm)	100A (12.7mm) O/P at 1 μ tesla (dBm)	100B (6.35mm) O/P at 1 μ tesla (dBm)
0.1	-54.80		
0.125	-52.86		
0.150	-51.28		
0.2	-48.78		
0.3	-45.26		
1.0	-34.80	-55.11	-78.73
3.0	-25.65		
6.78	-18.78	-38.97	
10.0	-15.62	-35.72	-60.32
20.0	-10.24		
27.120	-7.98	-27.45	
30.0	-7.26	-26.60	-51.32
54.240		-21.42	
100.0	-1.35	-16.71	-41.38
300.0	-2.39	-6.92	-32.16
1000.0		-0.94	-21.49
3000.0			-16.54

Table 2 Beehive 100C Frequency Compensation Chart

Plotting the data for the three antennae permits the selection of the correct loop for a particular operating frequency for the plasma under investigation. The operating points for each antenna are as follows:

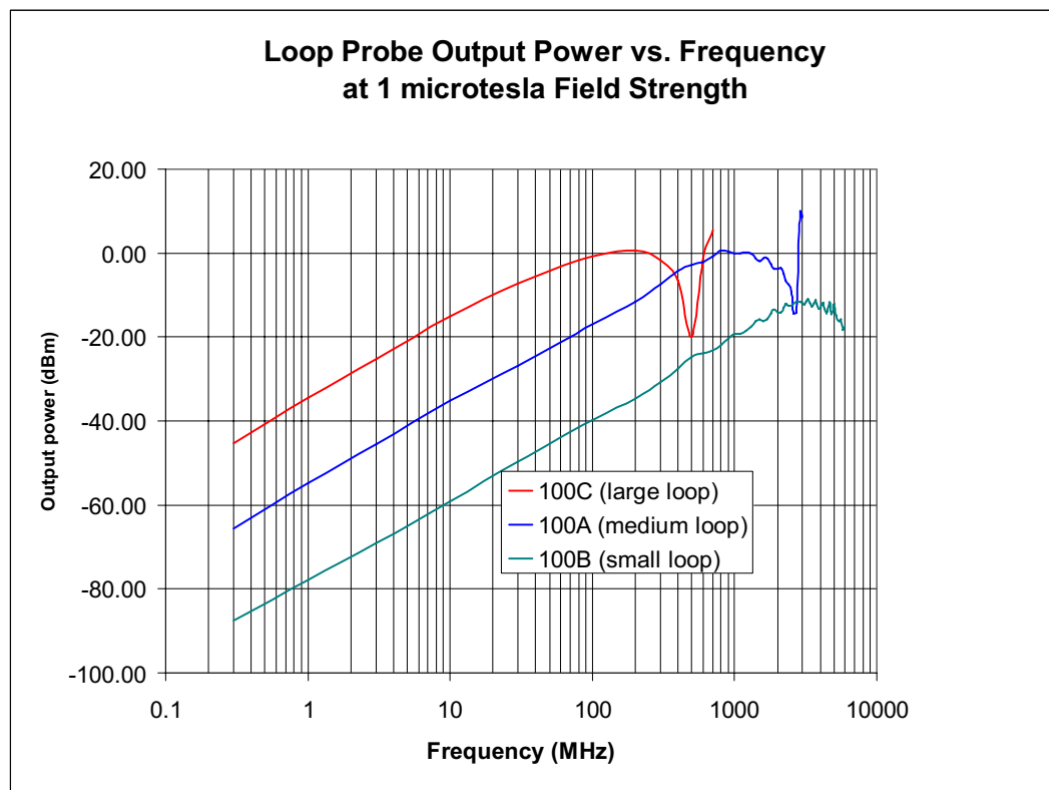


Figure 24 Loop Antennae Frequency Response (dBm vs MHz) (Beehive Electronics Inc., 2019)

The Beehive 100C is a 25 mm diameter loop and has frequency response up to 300 MHz. The correspondingly smaller loops 100A and 100B operate up to 1 GHz and 3 GHz, respectively. For the measurements made with the antenna external to the KF40 port such as in the figure (Figure 25) below, the Beehive 100C performed well.

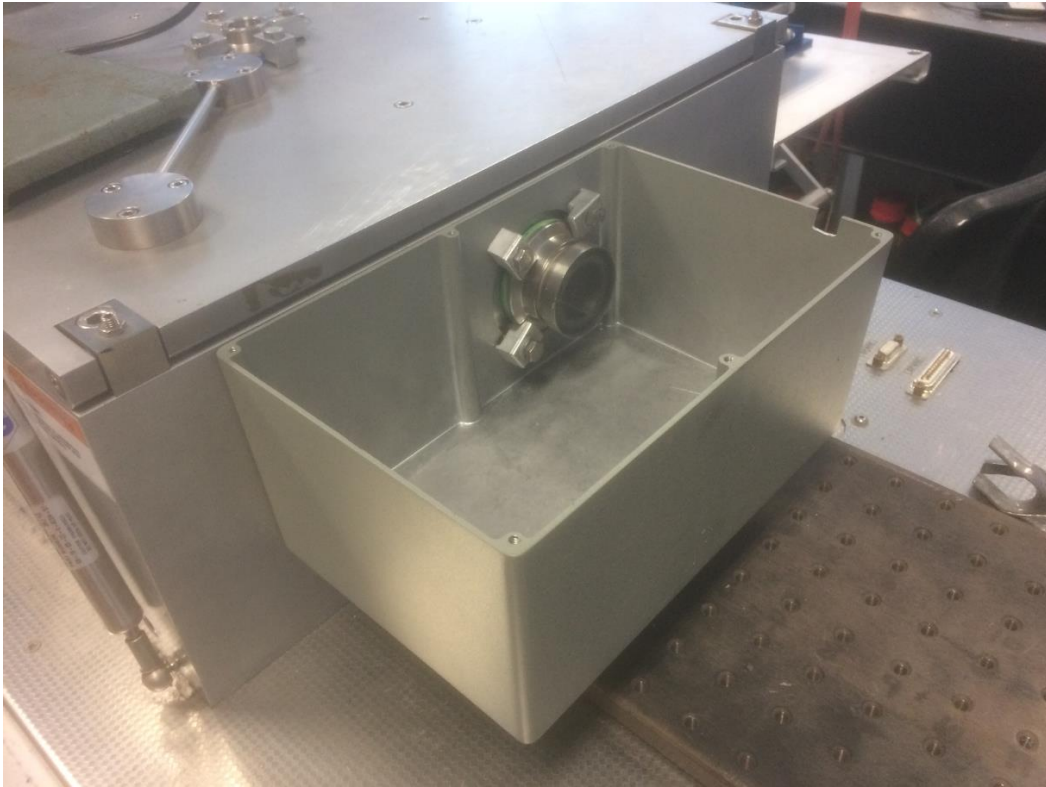


Figure 25 KF40 port with Quartz Glass and Location for Beehive 100C Antenna

It was found that the distance to the plasma was critical for the reception of high-order harmonics. Initial experiments for the TRES system for the detection of arcing events utilized the Beehive 100C located adjacent to the KF40 port. The other chamber was powered down so as not to interfere in the measurements. The RF screening enclosure in Figure 25 above reduced the noise picked up by the Beehive 100C. Additional work showed that the case of the tool was not RF grounded and significant levels of RF radiation were transferred by the machine panels to the unbalanced antenna cabling connected to the spectrum analyzer. Novel patent-pending designs to solve these problems are detailed in sections 3.10 and 3.11.

Later experiments for the Laplace modelling and the AI plasma classifications (see chapters 7 and 6) utilized a bespoke antenna that could fit inside the tube mounted

on the KF40 port so as to be flush to the inside of the chamber. This is shown in the two figures (Figure 26 and Figure 27) below:

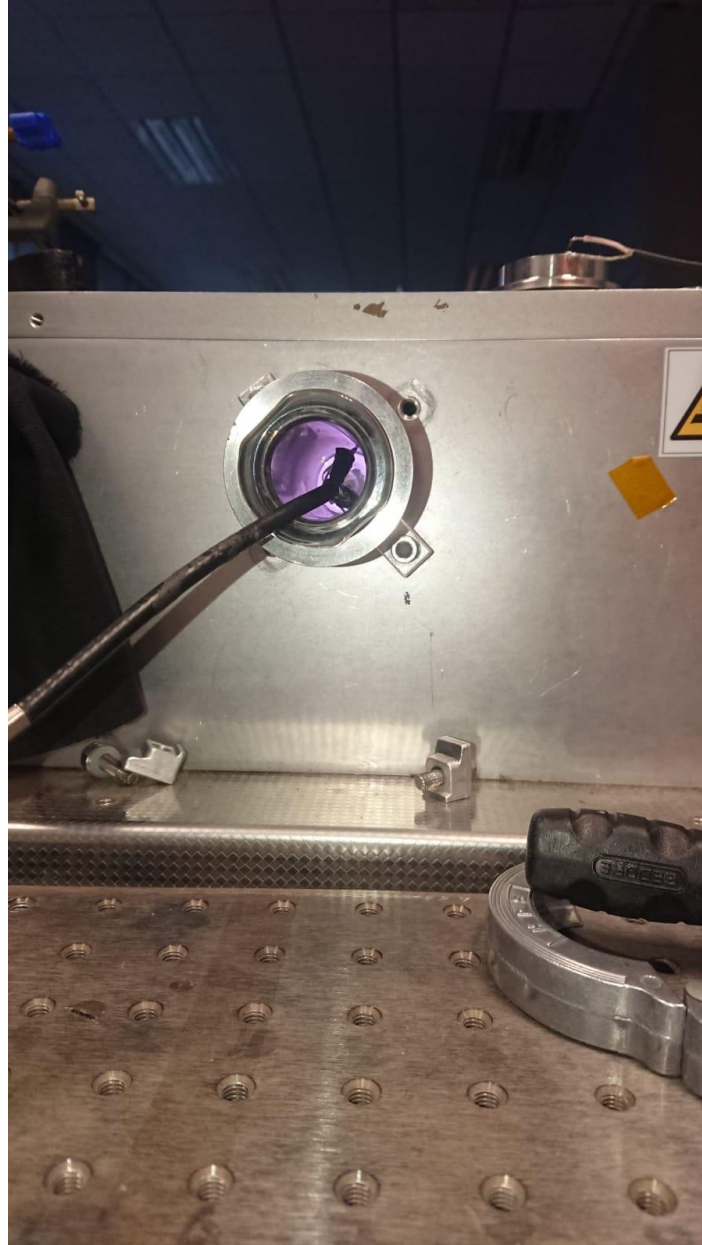


Figure 26 Location of Bespoke Plain-coil Antenna for FRES High-order Harmonic Acquisition

The location of the antenna flush to the chamber wall by inserting inside a test-tube through the KF40 port permitted the acquisition of harmonics up to nearly 1 GHz.

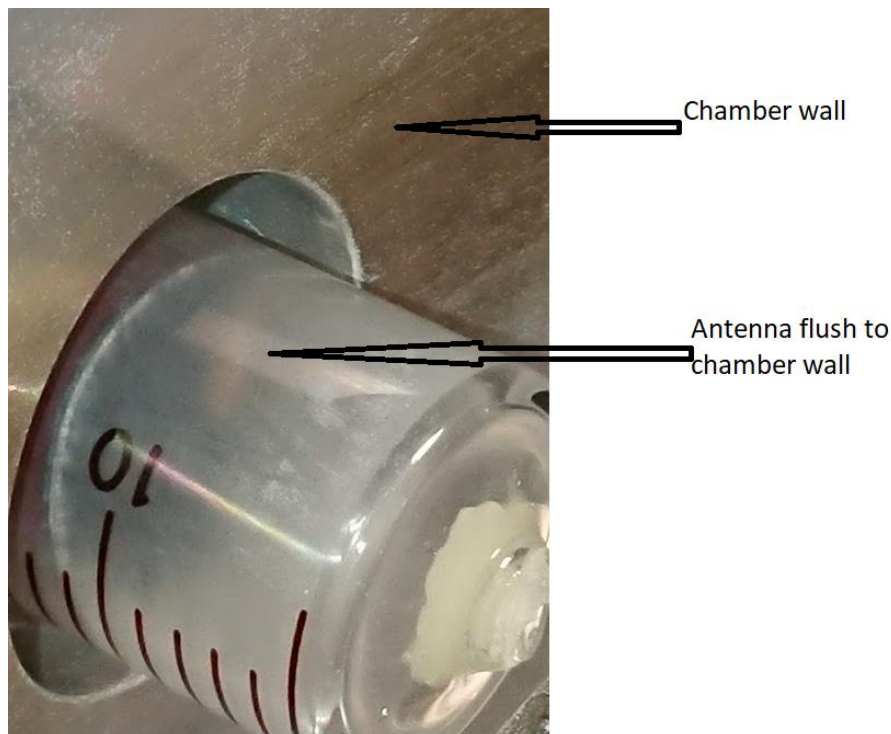


Figure 27 Location of Custom Plain-coil Antenna Flush to the Inside Chamber Wall for FRES high-order Harmonic Acquisition

This permitted a vector of over 100 harmonics to be used for data analysis. This will be covered in detail in Chapters 6 and 7.

3.8. Received Power and Current within the Plasma

The received power from the Beehive 100C antenna was mapped for 5 dBm increments and the corresponding voltages for driving a 50Ω load is shown in Table 3 below. As can be seen, the voltages delivered into the spectrum analyzer are very low and can be easily lost in the noise. The dynamic range of the amplifier and front end must have a low noise floor to register the signals at these levels. The SA44B has a very high dynamic range at the expense of a very narrow

bandwidth. Indeed, the components (Analog Devices AD9864) utilized in its design are for narrowband radio reception. The edge of the antenna probe was mounted directly up against the viewport of the Oxford Instruments PlasmaLab 100 CCP tool for maximum pickup. As noted, the outside flat surface of this port is located 245 mm from axial centre of the chuck.

dBm	Watts	Volts(rms)	Volts(p)	Volts(pp)
-120	0.100E-14	223.607 nV	316.180 nV	632.360 nV
-115	0.316E-14	0.398 μ V	0.562 μ V	1.125 μ V
-110	0.100E-13	0.707 μ V	1.000 μ V	2.000 μ V
-105	0.316E-13	1.257 μ V	1.778 μ V	3.556 μ V
-100	0.100E-12	2.236 μ V	3.162 μ V	6.324 μ V
-95	0.316E-12	3.976 μ V	5.623 μ V	11.245 μ V
-90	0.100E-11	7.071 μ V	9.998 μ V	19.997 μ V
...				

Table 3 Mapping from Received Signal Level to Plasma Current and Magnetic Field

The power out as measured by the antennae is as follows:

$$P_{out}(dBm) = X + 20 \log_{10}(B) + 20 \log_{10}(F) \text{ (Beehive Electronics, 2018)}$$

where:

B is the magnetic flux density in tesla

F is the frequency of the received signal in MHz

P_{out} is the probe output power into 50 Ω , in dBm

X is the scale factor from the Table (Table 4) below.

Model Number	X	3 dB Frequency (MHz)	First Resonance (MHz)
100C (large loop)	85.1	50	500
100A (medium loop)	65.2	1000	2600
100B (small loop)	42.2	3100	>6000

Table 4 Correction Factor Table for Power (Beehive Electronics Inc., 2019)

The frequency response of the Beehive 101A series (the 100C, the 100A and the 100B) is plotted in Figure 24 above. It worth noting that is it important to select the correct antenna for the frequency that is being emitted by the plasma and that each antenna has its own unique frequency response. They do not have a *flat* response by any means and when looking at harmonics of many multiples of the primary drive frequency, this must be taken into account in the calculations for the measured power.

It will be seen from the plots in the Figure 24 above that the antennae overlap one another in their useful frequency range. The Beehive 100C is useful for the Oxford Instruments PlasmaLab 100 CCP operating with its principal frequency of 13.56 MHz, but we note that for harmonics above approximately the 15th (approximately >200 MHz), the output is not calibrated and that the frequency is heading towards the first resonance of the antenna. This can be seen from the frequency plot above (Figure 24) where the output dips significantly at 500 MHz for the Beehive 100C antenna. Using the TRES and FRES sensor technologies, the use of the relative harmonics lends to a more robust measurement as the sensor is more tolerant to the absolute values. The absolute values are very dependent upon precise and repeatable location of the antenna system on the tool. This then requires a manual calibration and measurement operation to ensure the correct operation of the sensor. By using the relative harmonic measurements, some of the mechanical interdependencies of the sensor can be removed, easing adoption in an industrial environment.

3.9. Plasma Voltage and Current for Oxford Instruments PlasmaLab 100 CCP.

With the signals observed at approximately 245 mm from the axial central bulk of the plasma, we plot the recorded B- and E-field. This was recorded with the Rigol 500 MHz digital quad channel oscilloscope. This is for an Ar plasma with the pressure raised to a point that arcing events are taking place (A small piece of aluminium foil has been inserted in the chamber to force this arcing. The foil is shown in the figure (Figure 28) below:

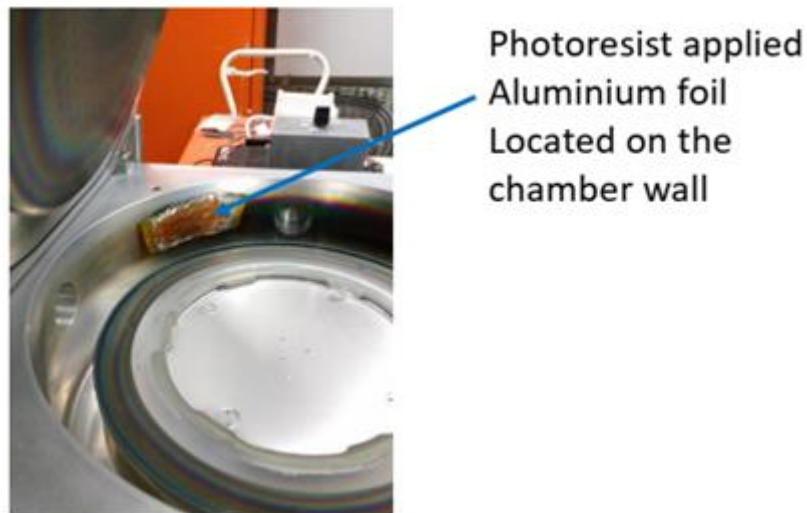


Figure 28 Operation of Oxford Instruments PlasmaLab 100 CCP Tool with Ar Plasma with AI Foil to Initiate Arcing

We will be examining this in more detail in Chapter 4 for perturbation detection systems.

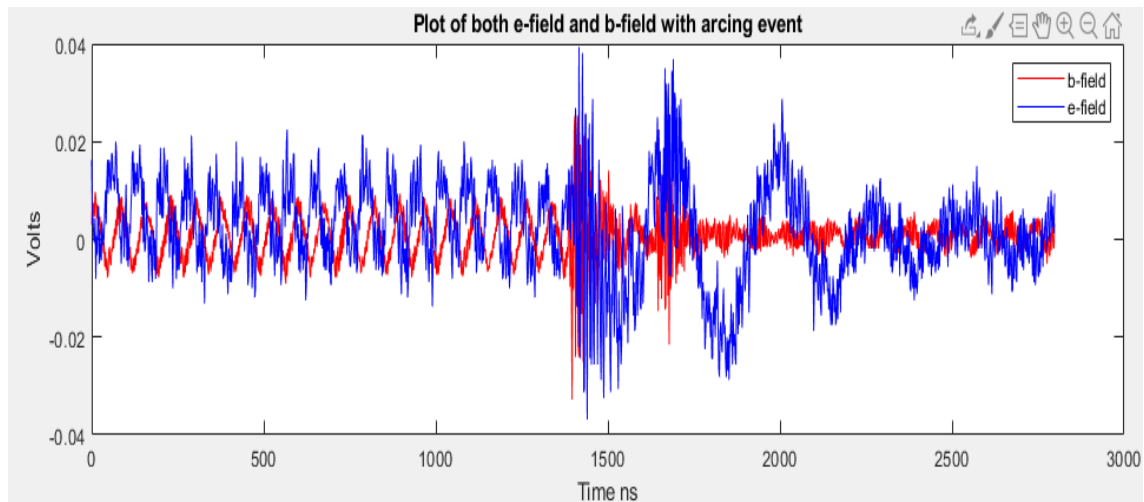


Figure 29 Recording of E- and B-field on Oxford Instruments PlasmaLab 100 CCP Tool using Rigol 500 MHz Oscilloscope

The initial recordings were made with the Beehive 100C for the B-field and the Beehive 100D for the E-field. We will later examine the signal-to-noise characteristics of the E- and B-field recordings for data analysis. For high-harmonic recordings with accurate harmonic absolute values, the single channel spectrum analyzer was a constraint. Having both E- and B-fields recorded and being able to phase resolve them, would be of great benefit. Multi-channel support is very much a requirement for future bespoke FRES hardware development.

3.10. Balanced Differential Antenna for Plasma Chamber.

The harmonics as received by the B- and E-field antennae are of a very low amplitude. During the experiments, the received fields were of the order of -100 dBm. The two chambers on the Oxford Instruments PlasmaLab 100 CCP tool have separate RF power generators and are approximately 500 Hz in frequency difference. To an extent, each chamber interfered with the monitoring of the other. It was very important to have isolation of the received signals from the chambers and from other outside sources. The antenna itself is balanced, but with its use of

a SMA connector and the subsequent cabling makes for an unbalanced line. The input to the spectrum analyzer is an unbalanced SMA connector. A patent-pending differential balun cabling system was designed to carry the signal from a RF isolated box surrounding the antenna to a remote mounted spectrum analyzer [“Electromagnetic field signal acquisition system for high signal-to-noise ratios, and electrical noise immunity”, GB Patent Application No. GB2005828.5. (Enterprise Ireland Identifier: 2020-DCU-001-PF). P.J. McNally, S. Kelly, D. Coates, *et al.* – filed 21 April 2020.]. In this remote mounted box, the signal was converted back to a single-ended cabling system for connection to the analyzer. This is shown in the figure (Figure 30) below.

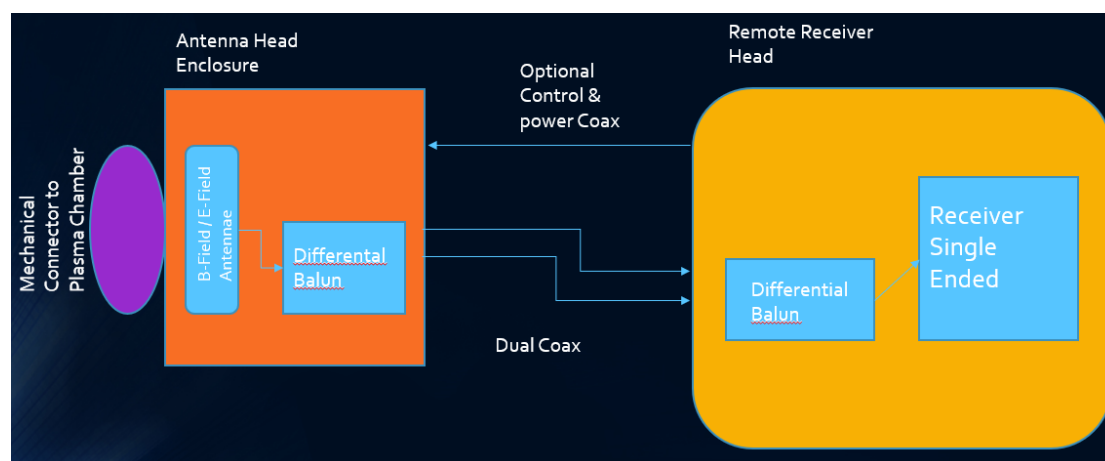


Figure 30 Differential Balun Isolated Cabling System

This permits the isolation of the cabling such that stray RF signals are not picked up from the adjacent chambers (common in an industrial environment) and also ambient radiation. During the experiments on the Oxford Instruments PlasmaLab 100 CCP tool in the DCU laboratory, a nearby mobile phone cell tower radiating at approximately 850 MHz was discernible in the recording of the harmonics.

Bespoke antennae with the custom differential balun system were designed, built and tested. The general arrangement of the antenna connection through the balun-cabling system up to the spectrum analyzer is show in Figure 31. The figure (Figure 32) below shows the mounting of one of the custom combined differential cabling and b-field antenna in the isolated box for mounting on the Oxford Instruments PlasmaLab 100 CCP tool. These antennae arrived from the PCB manufacturers after the experiments were completed for the digital signal processing (DSP) algorithms that were developed in the following chapters.

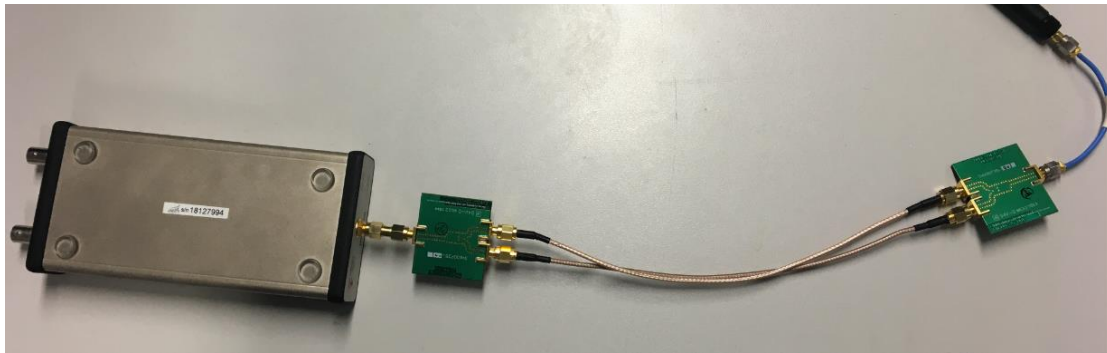


Figure 31 Differential Balun System Implementation

The figure (Figure 32) below shows the bespoke differential antenna mounted in the isolated box for connection to the KF40 flange on the Oxford Instruments PlasmaLab 100 CCP tool.

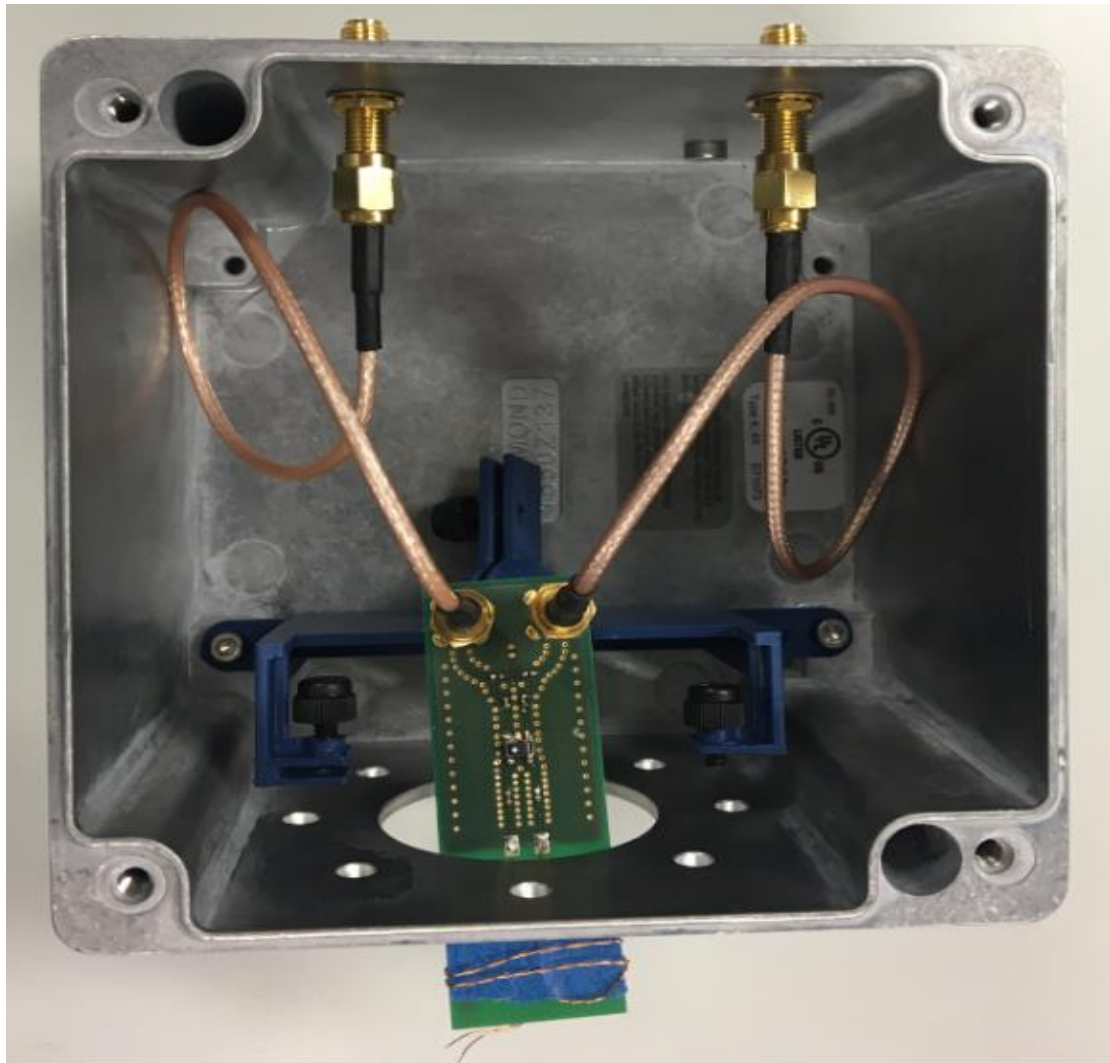


Figure 32 Bespoke Differential B-field Antenna in RF Shielded Chamber Enclosure (Dimensions of metal enclosure box = 110 mm X 120 mm X 120 mm)

This system permits the use of the FRES system in an industrial environment where multiple chambers are operating in close proximity to multiple RF sources and devices are operating on the unlicensed bands. Without accurate acquisition of the low-level harmonics within the plasma under observation, the concise measurement of the operating parameters would not be possible. The relative scales for the determination of the plasma parameters are the measurement of a signal of fraction of a dB at an absolute level of -100 dBm.

3.11. Companding (Compression/Expansion) Harmonic Signal System.

The dynamic range of the harmonic signals received by the FRES and TRES system from the plasma are of a relatively large order. With the Oxford Instruments PlasmaLab 100 CCP tool operating at 500 W, the received primary harmonic for the experimental antenna system mounted in the isolated RF enclosure was of the order of -22 dBm. The harmonics themselves are of a much lower amplitude with the lower harmonics descending to -105 dBm. We want as much dynamic range for each harmonic so as to detect changes in the operating parameters in the plasma. The high-order harmonics may only change by a few dB and we want to measure this change accurately. For a spectrum analyzer, the reference level is set to contain the highest-level signal and other signals received are then constrained by the dynamic range of the analyzer. We present a patent-pending companding system that compresses the dynamic range of the harmonics under observation to best match the dynamic range of the spectrum analyzer or zero-IF acquisition system for the FRES or TRES system.

1. The analogue-to-digital (AD) subsystem must be capable of acquiring the signals of the primary harmonics without distortion while at the same time being able to discriminate very small changes in the higher-order harmonics.
2. With a limited dynamic range, the low order harmonics will 'swamp' the bit-resolution of the AD subsystem and the higher-order harmonics will be lost in the noise.
3. We introduce a Plasma Characteristic ComPanding system that best matches the characteristic spectrum waveform. More usually this is to

suppress the low-order harmonics to equivalent levels of the high-order harmonics such that they are acquired accurately.

4. This plasma characteristic companding may be performed as a fixed standalone system or the response curve may be calculated to best reflect that of the observed harmonics and balance their powers to optimize the dynamic range of the acquisition system.

The figure (Figure 33) below shows the characteristic reception of harmonics on the Oxford Instruments PlasmaLab 100 CCP tool running an Ar plasma at 450W.

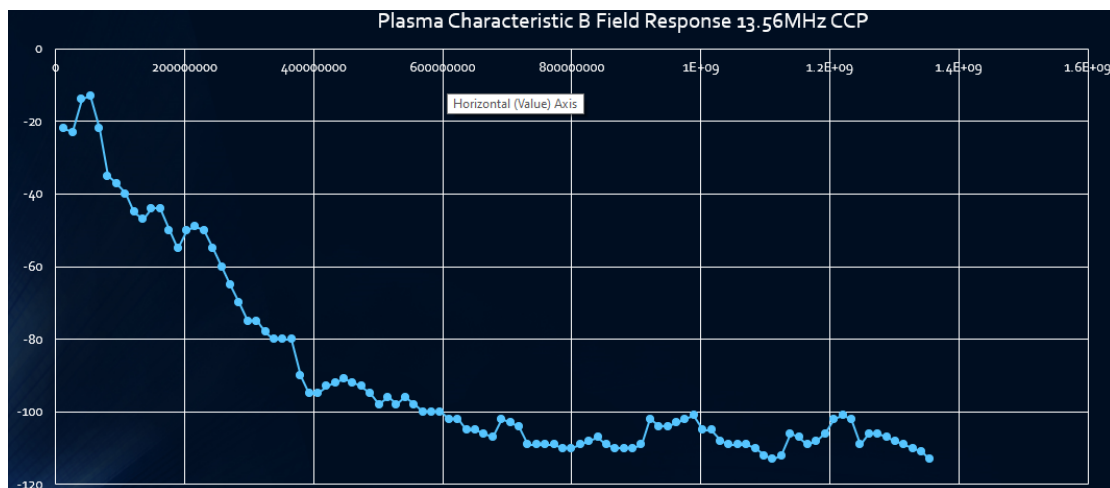


Figure 33 Characteristic Harmonic Signal Level for Oxford Instruments PlasmaLab 100 CCP Tool.

It can be seen in the figure (Figure 33) that the overall range for the received signals from the tool is approximately -15 dBm to -110 dBm. With a 16bit ADC giving a 96 dB range, the accuracy would be at best 1 dBm measurement. We will see later that the higher-order harmonics possess significant information for the detection of events that occur within the plasma. By applying a companding function the signal is dynamically reduced, but may be more accurately acquired by the radio subsystem.

The following diagram shows the transfer function of the companding system and how it compresses the dynamic range of the plasma harmonics.

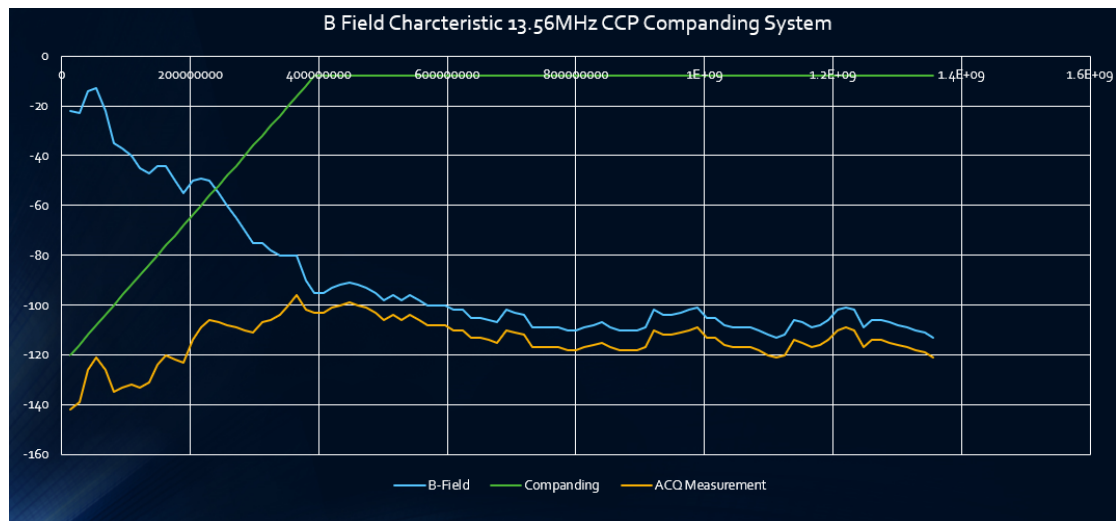


Figure 34 Companding Transfer Function and Subsequent Acquired Signal

The acquisition signal from the plasma chamber has been companded and now has a range -125 dBm to -95 dBm. In the case of the Oxford Instruments PlasmaLab 100 CCP tool, this may be as simple as a high-pass filter with a passband starting around 140 MHz. Each chamber and operating parameters with its antenna system would have to be characterized to model an optimum transfer function for the companding system.

Chapter 4 Arc Detection using Time Domain Analysis

4.1. Introduction

Plasma etching is of primary importance to the manufacture of semiconductor wafers. With the Oxford Instruments PlasmaLab 100 CCP, etching can take place with a number of reactive gases. A common fault occurrence is *arcing* within the chamber. This has the effect of damaging the wafers and is a very detrimental to yield. In dc sputtering systems, arcing is sometimes a problem due to patches of dirt (with higher secondary electron emission) (Chapman, 1980). These are classified as unipolar arcs (Maskrey & Dugdale, 1966) and may be removed by chamber conditioning. We examine a novel method to detect these arcs so as alert the operator and signify that the wafer may be damaged, before it continues to be processed and the error is only detected when the devices fail test schedules when the wafer is being functionally tested.

4.2. Time Domain Detectors

The main plasma chamber under test is the Oxford Instruments PlasmaLab 100 CCP system which uses a capacitively coupled 13.56 MHz RF source. Using an antennae hardware configuration similar to the FRES system, we record the radio spectra as emitted through the E- and B-field and analyze them. For the arc detection, we analyze in the time domain and in a continuous manner. For the TRES systems, we are able to use two of the quad channels of the Rigol digital oscilloscope to acquire the E- and B-field with the appropriate antennas co-located at the KF40 port. As shown, we require a pipeline to process the raw data from the time-domain collection. The events are very short lived and the analysis must be completed in real-time and in a continuous manner. We examine detection

methods to be applied to the continuous waveform as acquired from the B- and E-field to determine if arcs have occurred. We examine the *envelope* of the signal to detect changes in its characteristics. We will examine two detectors for envelope detection and we will also examine the use of the Wavelet Transform for event detection. In the case of the Oxford Instruments PlasmaLab 100 CCP tool, it is possible to see the collapse of the primary 13.56 MHz current source and its recovery when an arc event occurs. There is also the generation of a low frequency high E-field at the arc event.

The two main detectors under evaluation are the Square Law and the Hilbert Transform detectors. We will also apply the Wavelet Transform to detect instantaneous frequency shifts for arc detection.

4.2.1. Square Law Envelope Detector

The square law detector takes the original signal and uses it to demodulate itself (Roy & Roy, 2014, June). When squared, the resulting terms are in the order of the base signal plus some components that are multiples of the carrier. A simple low-pass filter after the square function blocks these terms and the resulting output is the envelope. With the loss of the primary drive signal in the Oxford Instruments PlasmaLab 100 CCP tool, this may be detected by the square law detector.

If we assign $y(t)$ as the primary (nominal, as the autotuner modulates the frequency for load matching) 13.56 MHz drive frequency then we are looking for the modulation of this signal to indicate arcing. If the modulation is expressed as:

$$y(t) = A_c(1 + mX(t))\sin \omega_t t$$

where:

$y(t)$ is the measured signal

A_c is the carrier signal

$mX(t)$ is the modulation of the carrier,

then the output of the square law detector $v(t)$ is:

$$v(t) = ay(t) + by^2(t)$$

where:

$y(t)$ is the signal detected which we combine with some fraction of the same signal squared.

If we multiply out the terms, we are left with a number of terms which are multiples of the carrier frequency and a single term that is the modulation itself.

This can be expressed as:

$$v(t) = ((\text{Carrier}) * (\text{terms})) + mbA_c^2x(t)$$

By low pass filtering this we can derive the amplitude modulation of the envelope of the (nominal) 13.56 MHz carrier. There are some scaling factors (m and b in the equation above) which can be accounted for but these are merely set in the threshold detection by setting an appropriate level for the threshold to be exceeded. The low-pass filtering will introduce a detection delay equal to half of the filter length. The pipeline for the square-law demodulator will also introduce some latency.

4.2.2. Hilbert Transform Detector

The Hilbert transform (HT) is a transform that converts a signal into an analytic one. An analytic signal is one that has no negative-frequency components. The analytic signal is a complex one where the real part is the original signal and the imaginary part is the Hilbert transform of the original signal (Matlab, 2018). It is a method of performing time-frequency analysis of switching transients. The basic function is to convolve the original signal with $1/\pi t$. This forces the integral in the convolution to diverge ($1/\pi t$ being a non-integral function). The Cauchy principle is applied in order to define the HT. This is just a method of assigning a value to the integral which would otherwise be undefined. Thus,

$$H(u)(t) = \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{u(\tau)}{t - \tau} d\tau$$

where:

$u(\tau)$ is the original signal

$H(u)(t)$ is Hilbert Transform of the signal

We can derive the envelope of the signal by evaluating the following:

$$y(t) = \sqrt{x(t)^2 + h(t)^2}$$

where $h(t)$ is the Hilbert Transform of $x(t)$ (Ulrich, 2006, March, Los Alamos National Laboratory). Any real sinusoid $A \cos(\omega t + \phi)$ may be converted to a positive-frequency complex sinusoid $A \exp[j(\omega t + \phi)]$ by simply generating a phase-quadrature component $A \sin(\omega t + \phi)$ to serve as the imaginary part:

$$Ae^{j(\omega t + \phi)} = A \cos(\omega t + \phi) + j A \sin(\omega t + \phi)$$

This phase quadrature component can be generated from the in-phase component by a simple quarter-cycle shift. For more complicated signals as we have from the Oxford Instruments PlasmaLab 100 CCP (at least 5 harmonics of the base 13.56 MHz plasma drive), filters can extract each of these and they can be shifted by a quarter cycle. This is the Hilbert transform of the direct signals from the loop antenna. If the filter has magnitude of '1' at all frequencies and introduces a phase shift of $-\pi/2$ at each positive frequency and $+\pi/2$ at each negative frequency then the real signal $x(t)$ and its Hilbert transform $h(t) = H_t\{x\}$ are used to form a new complex signal of the form $y(t) = x(t) + j y(t)$, the signal $y(t)$ is the (complex) analytic signal corresponding to the real signal $x(t)$. This means that the analytic signal of $x(t)$ has the property that all 'negative frequencies' have been 'filtered out'. With this we can derive the envelope and detect when the arcing is occurring.

4.2.3. Wavelet Transform Detector

A wavelet describes a function that begins and ends at zero. There are a number of basic ones that satisfy a number of properties that makes them useful for signal processing and in our particular case, transient detection. They are particularly useful when the signal has discontinuities and sharp spikes (Lee, Libedinsky, Guan, & So, 2017 May). Whereas the Fourier transform is the approximation using the superimposition of continuous sine and cosine waves, wavelet analysis looks at the signal with a varying scale and applies discrete wavelets. Wavelet algorithms process data at different scales or resolutions. By looking at a large window of signal, we could determine the base features. If on the other hand we

look at the signal with a small window we would notice 'small features'. Fourier transformation makes use of continuous infinite signals and are non-local, but by their very definition, transients are local. Therefore, quite often, they do a very poor job in approximating sharp spikes. With the wavelet analysis, we can use approximating functions that are contained neatly in finite domains. The approximations of data with sharp discontinuities are well suited for wavelet analysis (Graps, 1995). The choice of wavelet is probably the single most important factor. The other two main factors are Scaling and Shifting. Some of the available wavelets are shown below. Each of the wavelets has a specific characteristic that lends itself to a particular type of data that is to be analyzed. See Figure 35 below for some of the different wavelets.

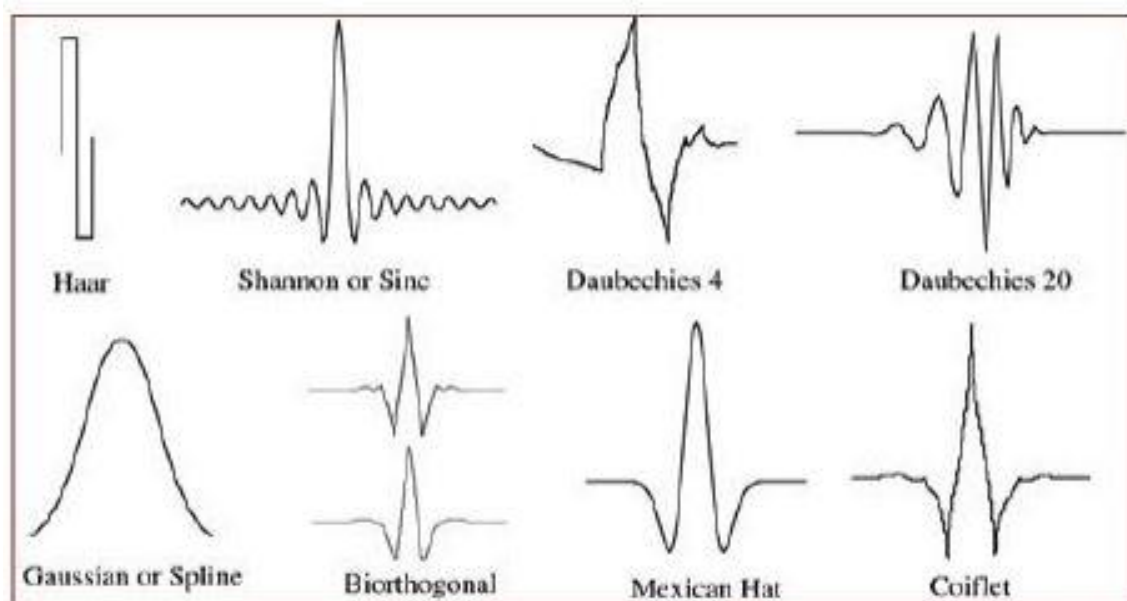


Figure 35 Basic Wavelets for Transforms (Andrew Tan, 2016)

For this analysis the basic wavelet used is the Morse Wavelet. The Morse wavelets are complex valued whose Fourier transforms are supported only on the real positive axis (Matlab, 2018). The figure (Figure 36) below shows the basic Morse

wavelet as used in our Matlab® analysis of the Oxford Instruments PlasmaLab 100 CCP tool arcing experiments.

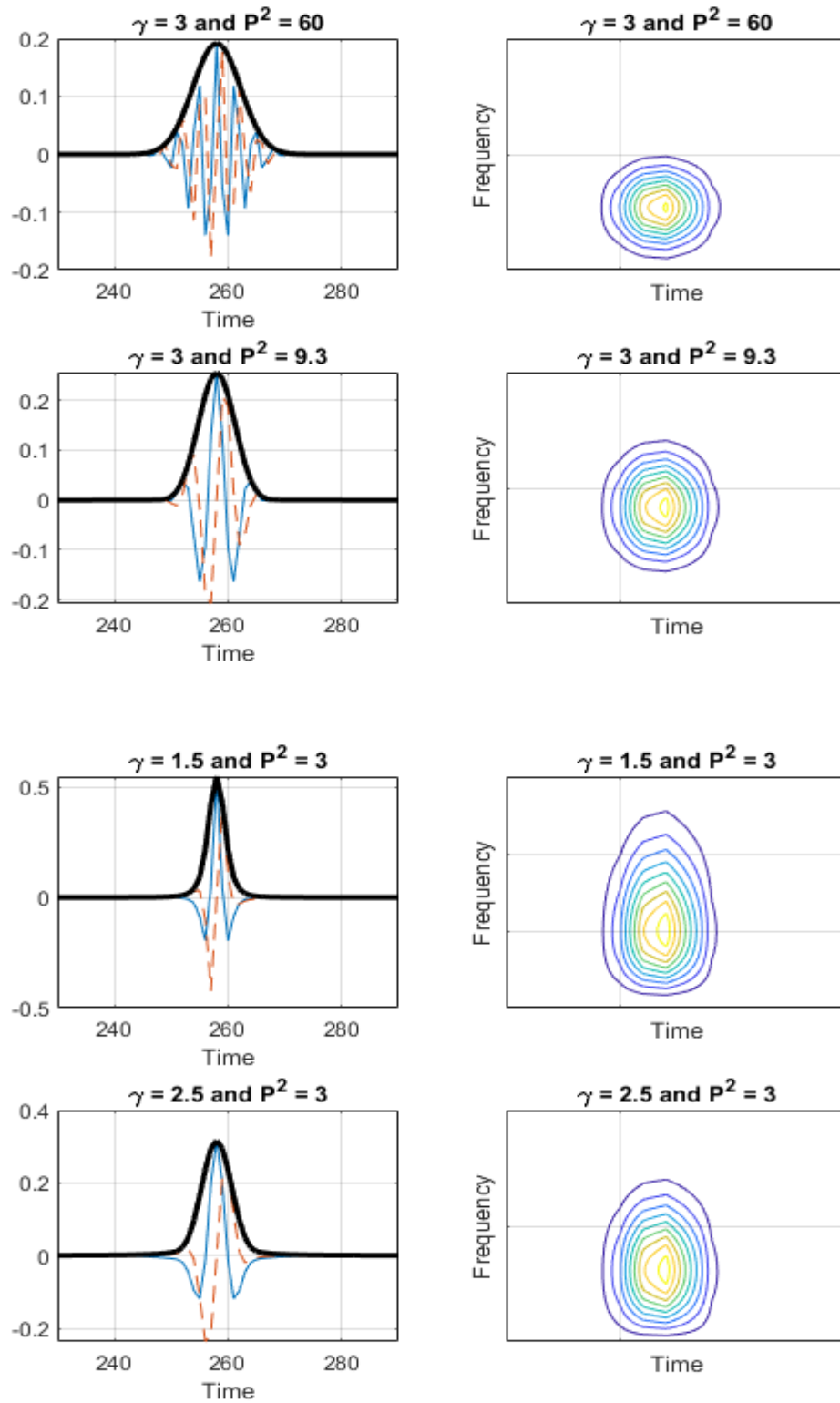


Figure 36 Morse Wavelet with Different Time-Bandwidths (Matlab, 2018)

This makes them useful for analyzing modulated signals with time-varying amplitude and frequency. The Fourier transform of the generalized Morse wavelet is:

$$F_{p,y}(\omega) = U(\omega)a_{p,y}\omega^{\frac{p^2}{y}}\exp(-\omega^y)$$

The value of $U(\omega)$ is the unit step, the value $a_{p,y}$ is a constant. The p^2 value determines the spatial spreading of the signal and so determines the frequency of the resultant wavelength. For a range of wavelets p^2 is replaced by βy .

The Wavelet Transform is a time-frequency analysis representation that is suitable for non-stationary signals. As described above, the Wavelet Transform uses a large library of wavelet functions having different characteristics with variable window length as opposed to the Fourier Transform which uses sine and cosine functions for the analysis using a fixed window size. (Note that there are other methods to use short term variable window sizes for the Fourier transform, such as Hamming windows which can concentrate frequency components around short term temporal events. The short-term Fourier transform gives uniform resolution in the frequency domain, but this is not ideal for the detection of high-speed transients). The Fourier basis functions extend infinitely along time whereas the duration of the transients of interest are short with respect to the observation interval. The narrowband signals can be detected from the ambient noise, but the transient energy events, while they do effect the Fourier spectrum of the signal under analysis that determines the amount of change, depend upon the ratio of the transient energy to noise energy and the narrowband energy

(Learned & Willsky, 1993) . If the ratio is quite large with respect to the time or frequency axis, they may be difficult to detect. The time-frequency expansion of the plasma B-field signal can be implemented via orthogonal wavelet transforms as we are interested in the preservation of the energy in the analysis stage (Choosing a Wavelet) (Matlab, 2018). These decompose the observed signal into a succession of orthogonal subspaces at different scales. This *decomposition* is performed by using a single prototype function or analyzing wavelet. For accurate frequency analyses within the observed signal, the analyzing wavelet is expanded in time and for accurate instantaneous frequency analyses the analyzing wavelet is compressed (Lilly & Olhede, 2011,Oct). The Fourier spectrum of a signal (x) is given by $X(f)$ where,

$$X(f) = \int_{-\infty}^{+\infty} x(t) \exp(-j2\pi ft) dt$$

This is the continuous Fourier Transform. The Inverse Fourier Transform function returns $x(t)$ for $X(f)$. Since the transient event is but one, it is impossible to find the time of its occurrence over the range $\pm\infty$. It is possible to *window* the signal $x(t)$ before calculating the Fourier Transform. This called the short-time Fourier transform (STFT) and thus continuous Fourier transforms are performed on the signal for individual instants of time. The choice of *windowing* function becomes important for different views of the transformed signal. We therefore create a *windowing* function that defines a time window for the Fourier analyses. The sliding window Fourier Transform is as follows:

$$S_x(t, f) = \int_{-\infty}^{+\infty} x(\tau) \omega(t - \tau) \exp(-j2\pi f\tau) d\tau$$

The sliding windowing function is $\omega(t)$. Note that the multiplication of the input signal and the window function in the time domain corresponds to a convolution in the frequency domain. For example, in the frequency domain, the magnitude of the transfer function of the rectangular window is a sinc function. Due to convolution in the frequency domain, the resulting signal spectrum is smeared, and the resultant spectrum covers wider frequencies (Rauscher, Christopher, 2014). There are various shapes for the windowing function: common ones are the Gaussian, the rectangular (which introduces nulls and side-lobes as above), and the Hann window (which introduces amplitude errors (University Lowell Massachusetts , 2019)). Optimized windows are generally used instead of rectangular windows as they exhibit lower secondary maxima in the frequency domain and the leakage effect is reduced. Flat top windows produce a high level of accuracy, but produce a wide main lobe which reduces frequency resolution. The window based on a Gaussian window is called a Gabor transform (Gabor, 1944).

4.3. Oxford Instruments PlasmaLab 100 CCP Tool Arcing

The Oxford Instruments PlasmaLab 100 CCP tool in DCU can also be used as a source of plasma arcing. By raising the pressure significantly, arcing can be induced within the chamber. We can see this on the oscilloscope acquisition in the diagram below. Analysis of the E- and B-field signals shows the subsidence of the primary RF drive current component with high frequency components followed by the re-establishment of the primary drive frequency. At the same time, the electric field increases at the arc event. As we shall see later, the B-field was a more reliable indicator of arcing events.

4.4. Simple Threshold Detector for Oxford 100 Arcing

By raising the pressure within the Oxford Instruments PlasmaLab 100 CCP chamber in an Argon plasma at 100W arcing occurred regularly on a piece of aluminium foil placed by the chuck. See Figure 37 below.

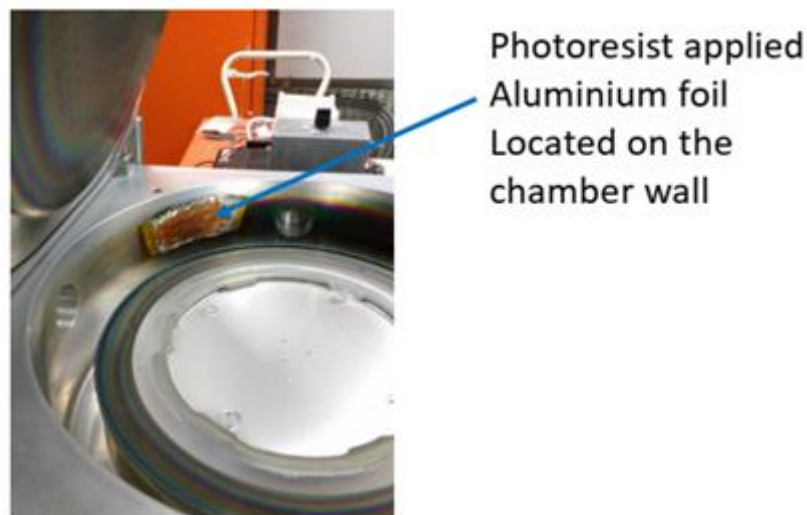


Figure 37 Foil Used to Initiate Arcing Events with High Pressure in the Oxford Instruments PlasmaLab 100 CPP Tool Chamber

The frequency of arcing events could be seen quite clearly through the observation window at approximately a 1 Hz rate. The near-field B-field acquisition was recorded using a multi-channel digital oscilloscope. The oscilloscope utilized for this experiment was a Rigol DS6104 which has four acquisition channels each with a 1 GHz bandwidth and a low-level signal detection down to 20 mV per division. By using the one-shot threshold triggering mechanism, the signal was acquired at a 500 MHz rate. An initial examination of the time-series data shows a lowering of the fundamental 13.56 MHz RF primary B-field amplitude and the generation of a low frequency high amplitude E-field. To test this, a software low-pass filter was designed with a cutoff of 20 MHz (primary RF is at 13.56 MHz) and an order of 400

(i.e. the digital filter is expressed with 400 coefficients to give a very sharp falloff in the stop-band) with the sampling frequency of 500 MHz (as this matches the sampling frequency of the acquired samples from the Rigol oscilloscope).



Figure 38 Screenshot of Real time E- and B-field Acquisition on the Oxford Instruments PlasmaLab 100 CCP - Forced Arcing. The B-field signal is shown in blue; the E-field signal is shown in yellow.

The timed E- and B-field recordings stored on the oscilloscope were then transferred to USB memory for further processing. They were recorded as comma separated values (CSV) files. Note that the timebase is the same for both recordings. Matlab® functions were then written (see Appendix F) to extract the records and the following plots were produced for the E- and B-field. The time base was expanded to show clearly the arcing event as the B- and E-field were recorded on the two channels of the oscilloscope. There are approximately 1350 sample points at a timebase of 500 MHz (2 ns per division).

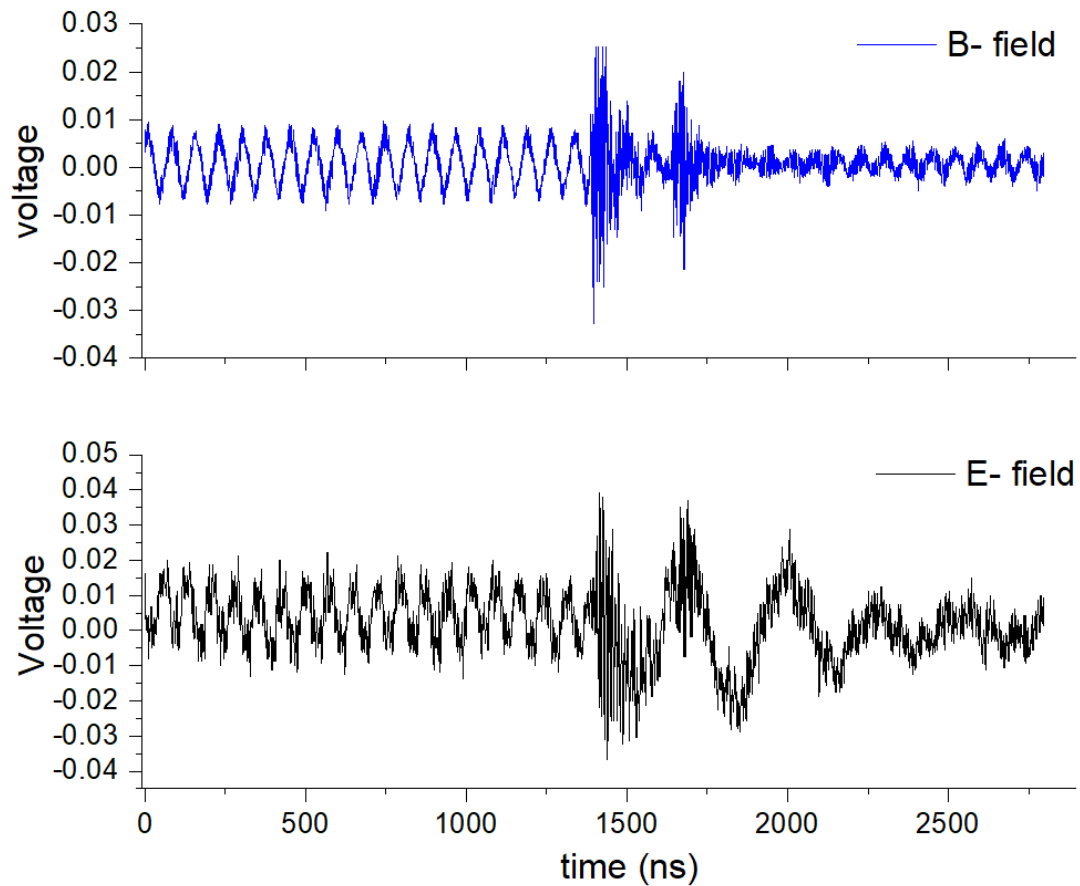


Figure 39 Recorded E- and B-fields. Oxford Instruments PlasmaLab 100 CCP Forced Arcing with a Single Event at 1300 ns

These recordings were then utilized in the Matlab® filters as the raw data for analysis. First, we plot the normalized filter response. It is shown in the Figure 40 below.

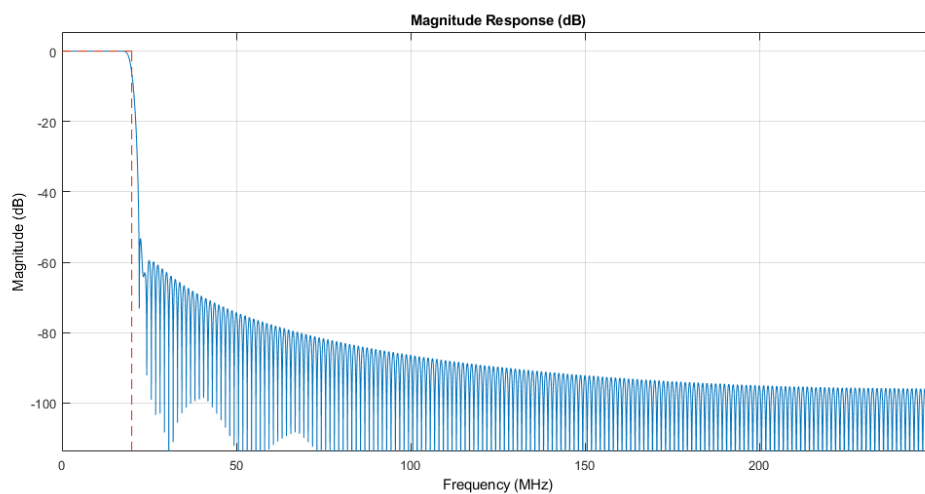


Figure 40 High Order Low Pass Filter Normalized Response for LPF Detector

The coefficients of a digital filter determine the transfer function and the specification of the filter's poles and zeroes. As a single pole filter gives 6 dB per octave and a 2-pole filter gives 12 dB per octave, the high order (400 in this case) gives a sharp knee. By applying the B-field and then the E-field CSV data (representing the samples acquired on the Rigol oscilloscope for the arcing event) to this filter, this shows the very clear primary RF drive. It can be seen in the processed output of the filter (see Figure 41 below) that the primary B-field RF 13.56 MHz as recorded, collapses as the arc event occurs. A simple threshold detector applied to the output of the filter can then be used to determine arcing events. The threshold can be set by equating it to the set recipe-required power. Note that this system relies on the specific driving capabilities of the Oxford Instruments PlasmaLab100 CCP tool and is not a general system with reference to power levels and frequency constraints. To make it a more general detection subsystem would require adjustment of these parameters for the particular machine under examination.

Applying this filter to the recorded signal gives the following response. It can be seen that the simple threshold detector will work in this situation for the Oxford Instruments PlasmaLab 100 CCP tool.

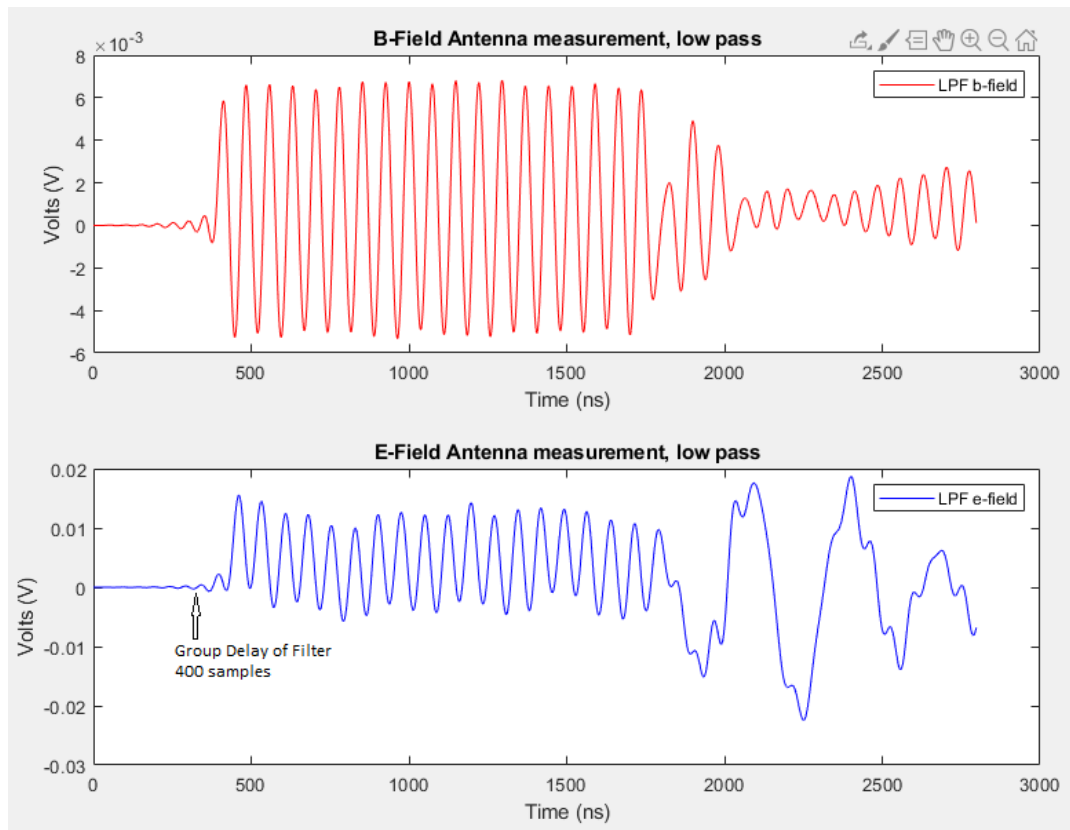


Figure 41 Arcing Event with LPF Response on TRES E- & B-Fields for Arc Detection (Primary 13.56 MHz)

The outputs of the LPF filters are the 13.56 MHz B-field and the E-field data. This has a period of ~ 73.7 ns which we can see plotted as 6.8 transitions in 500 ns. As can be seen, the primary drive current decreased rapidly within a few cycles at the arc event and the electric field increased with a low frequency sinusoid being dominant. At the 1700 ns point we see the first part of the arc event where the B-field is collapsing and later at 2000 ns we see the most significant collapse of the 13.56 MHz B-field. (Note that these times are delayed by the group delay of the filter. The group delay is due to the order of the digital filter (400 samples in this analysis from time of input (0) to time of filter output) as seen in the graphed response). The B-field then begins to recover from about 2300 ns point. Further investigation is required of the recovery section (from location 2000 ns onwards) by comparing it to the output of the Oxford Instruments PlasmaLab 100 amplifier

RF drive and the autotuner section. For an arc event signalling, setting a threshold at ± 4 mV for a peak threshold detector of the output of the B-field LPF and inverting the response would permit the detection. This detector registers an event detection in 400 clock cycles at 500 MHz giving a detection in 800 ns or approximately 11 RF cycles. The E-field LPF could be monitored for event excursions, but the signal, in general, contains significantly more noise as can be seen in Figure 39 above. For increased accuracy, the two signals could be logically 'anded' to determine that an arc event has occurred. We now look to more concise determination of the arcing event and its classification.

4.5. Square Law Envelope Detector for Arc Detection

The LPF arc event detection as described in the previous section is tightly tailored to the working parameters of the Oxford Instruments PlasmaLab 100 CCP tool. By designing a high order filter specifically for the RF drive frequencies as utilized in the tool, the method is functionally correct but not very adaptable. Should other drive frequencies be utilized in a plasma tool, the filter would require updating to match this. If the design was to examine the waveform for event detection of the natural plasma frequency, it would be advantageous if the detection method was independent of the applied frequency and the filters did not require fixed values. By using self-decoding demodulators, it is possible to remove the requirement for fixed parameter filters. The square law detector is such a demodulator. The output filter of the square-law envelope detector is set below any fundamentals that may appear in the observed signal. In this case (Oxford Instruments PlasmaLab 100 CCP tool with 13.56 MHz primary RF drive) a filter with a cutoff of 10 MHz and with an order of 400 is designed using the Matlab toolkit. This toolkit permits the

rapid development of digital finite impulse response (FIR) filters where the characteristic parameters are known (sampling frequency, flatness in the passband, cutoff frequency and flatness in the stopband). We chose a high order steep cutoff with a sampling frequency of 500 MHz to match the input samples that we have recorded for the arc events on the Oxford Instruments PlasmaLab 100 CCP tool. The response characteristics for this filter are shown as follows. See Figure 42 below.

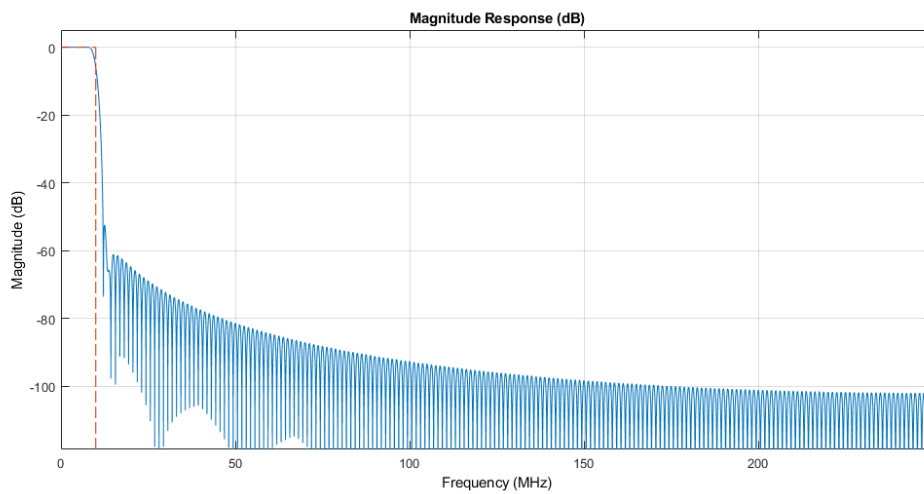


Figure 42 Low Pass Filter Normalized Response (10 MHz cutoff, Order 400, $F_s = 500$ MHz) for Square-Law Envelope Detector

The output of the decoder is as shown in the figures (Figure 43) that follow. The code for the square-law detector is given in Appendix F. The detected envelope is shown plotted over the sampled E- and B-field acquisition and it can be seen that a simple peak-detector will show the occurrence of an arcing event. We can see that the arc event is recorded in the B-field at the 1350 ns point. There is a large high-frequency burst recorded in the B-field followed by another burst at 1600 ns. For the E-field, there is large amplitude low-frequency electric field recorded in the measurements at the arc-event. The data is low pass filtered before being applied to the square law envelope demodulator and we can see this in the

measured response. (The output is delayed by the group delay of the LPF, 400 samples). The two plots are overlaid with the output of the envelope demodulator shown atop the E- and B-field arc event. See Figure 43 below.

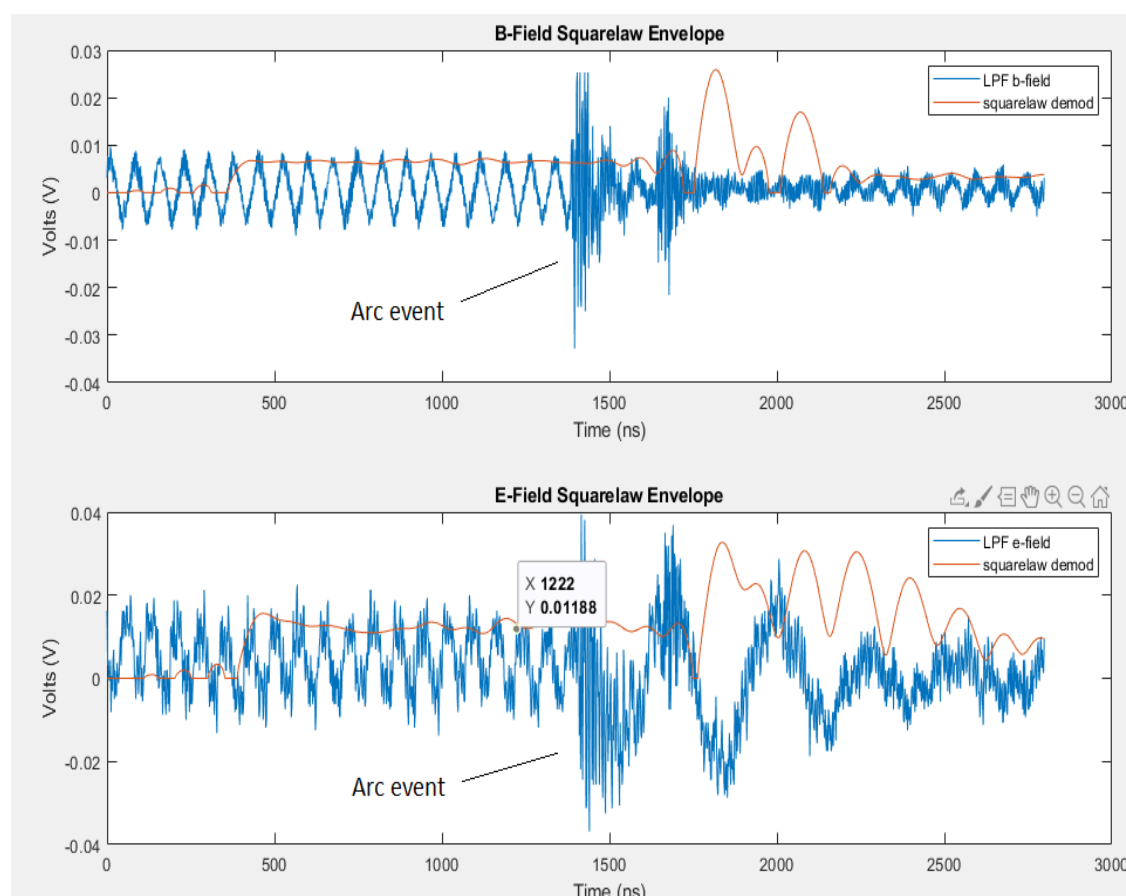


Figure 43 Square Law Detector, blue=(E-field, B-field), red=Squarelaw (B-field, E-field)

With the arc it would seem that the autotuner deposits its energy into the plasma and the B-field peaks indicate a sharp current spike. The electric field increases sharply after this current spike but with a low frequency of the order of 3.3 MHz. The primary RF current falls off and then slowly recovers. No corresponding measurement was made of the output of the RF amplifier driver or the output of the auto-tuner. Further work could look at the energy transfer in order to examine whether it is in the plasma itself (high Q) or whether it is in the auto-tuner or

amplifier output. The demodulated output may be used as a detector for arcing events. There is a large current transient and this could be used to trigger a threshold circuit. The low-frequency E-field sinusoid may be a characteristic of the auto-tuner circuit and the plasma and its specific frequency may not be relied upon as a detector. The high E-field transient could be utilized in a threshold detector. The two demodulated outputs could be combined for a more accurate determination of an arcing event.

4.6. Hilbert Transform Detection of Arcing Events

The normalized Fourier Transform of the first half of the signal is shown below in Figure 44, with the primary 13.56 MHz drive RF being evident. We divide the recording into two separate components. We examine the pre-arc event and then the complete recording which includes the arc event. We take the Fourier Transform of the data to a location just before the arc event occurs (i.e. at 1350 ns). Therefore, by selecting the data before this sample point, we can determine the frequency components of the B- and E-field of the normal operation (no arcing) of the plasma. We then take a second Fourier Transform of the E- and B-field, but this time including the arc event and we overlay plot the two frequency data sets. See Figure 44 and Figure 45 below. The plot indicates that the density of the high frequency components increases but note that it does not contain a time reference for this occurrence. We were also cognizant of the arc event such that we could perform a 'before' and 'after' Fourier Transform to compare the spectral densities. This is shown below for the B- and E-field measurements. It is noted that the E-field is significantly more noisy than the equivalent B-field measurement,

despite many attempts with RF shielding and cabling to reduce the noise. The plasma itself and the location of the antenna seems to give rise to this noise. Later, in chapter 6 and 7 with the use of the FRES measurements for the signal analysis acquired through the SA44B and its single channel, the B-field was chosen for acquisition due to its more coherent harmonic content.

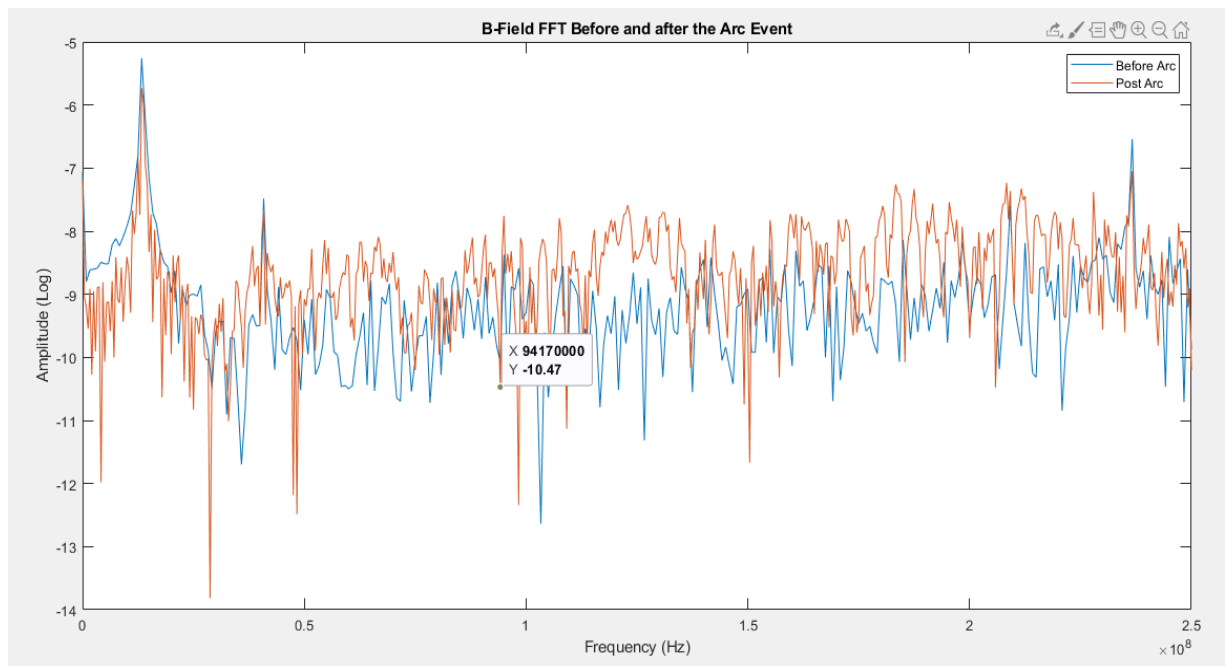


Figure 44 Relative Fourier Transforms of the B-Field Acquisition before Arcing Event (blue) and across Arcing Event (red).

The equivalent E-field pre- and post-arc Fourier Transforms showing the movement of power from the lower frequencies to the higher frequencies is plotted.

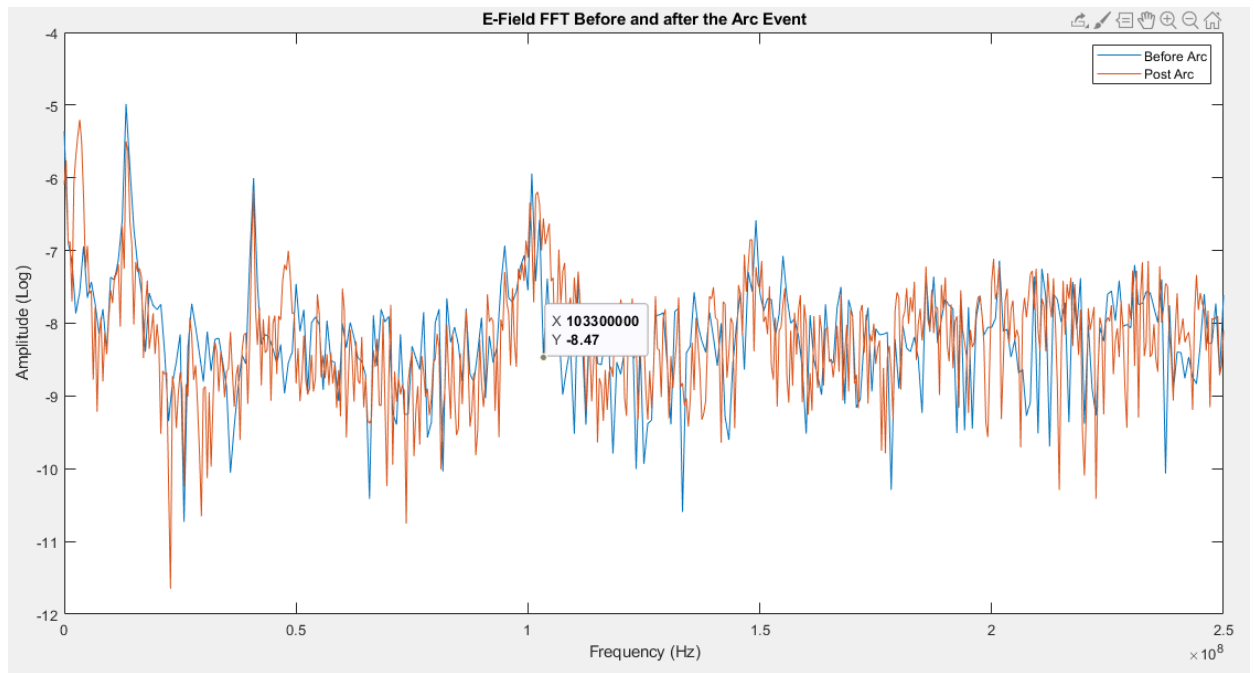


Figure 45 Relative Fourier Transforms of the E-Field Acquisition before Arcing Event (blue) and across Arcing Event (red).

There is a marked contrast between the B- and E-fields before and after the arcing event. While they both show only a 1.5 dB drop in the primary drive RF frequency, the E-field data contains relatively large amounts of high-frequency noise both before and after the arc event. The large low-frequency electric field at 3.3 MHz is the dominant change in the measured electric field.

For the B-field measurement, it can be seen that the relative noise floor has increased with wideband noise and that the distinguished harmonics intensities are lowered. By taking the Hilbert Transform, the plot of the real component of the signal delivers the same signal (the original signal being a real signal). The code for generating the values have been calculated by using the Matlab toolbox functions and also by using basic direct calculation methods on the matrix arrays to calculate the coefficients (see Appendix G). It is possible to perform the calculation by taking the Fourier Transform and then rotating the real and

imaginary coefficients appropriately or using the direct Hilbert Transform toolbox. In the basic method, the Fourier Transform of the sample data is calculated and the positive frequency coefficients are rotated counter clockwise and the negative frequencies are rotated clockwise. The inverse transform is then calculated from this matrix. The code is also included in Appendix G. By taking the magnitude of the Hilbert Transform and squaring this, it is possible to measure the power (Jungwirth, 2016,Dec). The time-frequency component is observed when the phase angle component of the Hilbert Transform is plotted. When we examine this phase angle component, we see that the arc event is detectable.

In Figure 46 below, we calculate the Hilbert Transform (HT) of the recorded B-field samples of the B-field arc event and display that in the first subplot. The second sub-plot is that of the real component of the HT showing the reconstruction of the original signal. The third subplot is a plot of the phase angle of the HT. This shows the instantaneous phase of the signal. We note the change to high frequency energy as the instantaneous phase is changing rapidly.

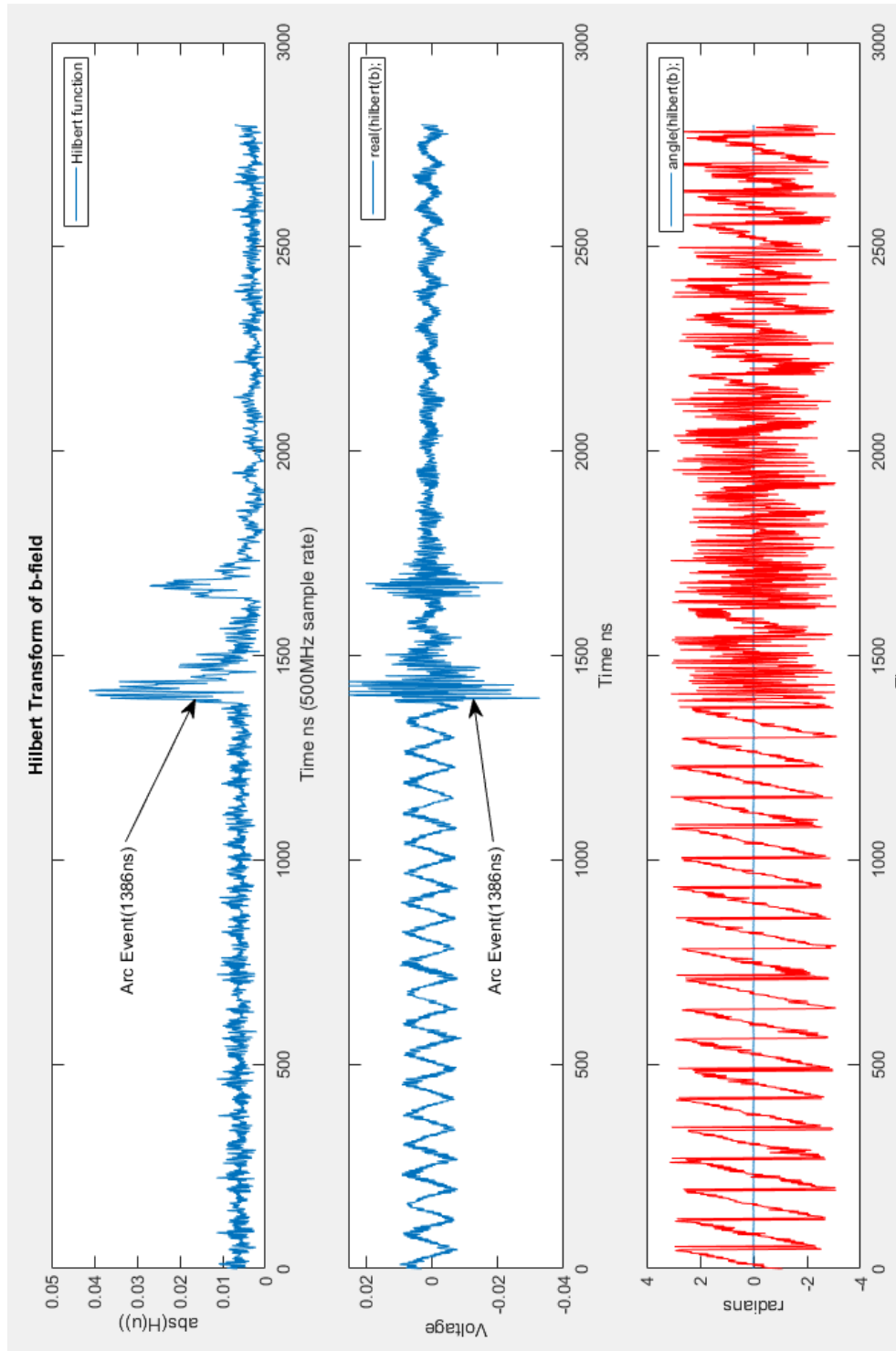


Figure 46 Hilbert Transform of B-field, Real Component of HT and Instantaneous Phase Angle of HT of the B-field

The instantaneous phase angle can be measured and its rate of change can be used as a detector for arcing events. The magnitude of the Hilbert Transform could also

be used as an event detector, but this is similar in operation to the envelope detector described.

This calculation is then repeated for the E-field. The following plot is obtained when the Hilbert Transform is applied to the E-field recording with an arc-event.

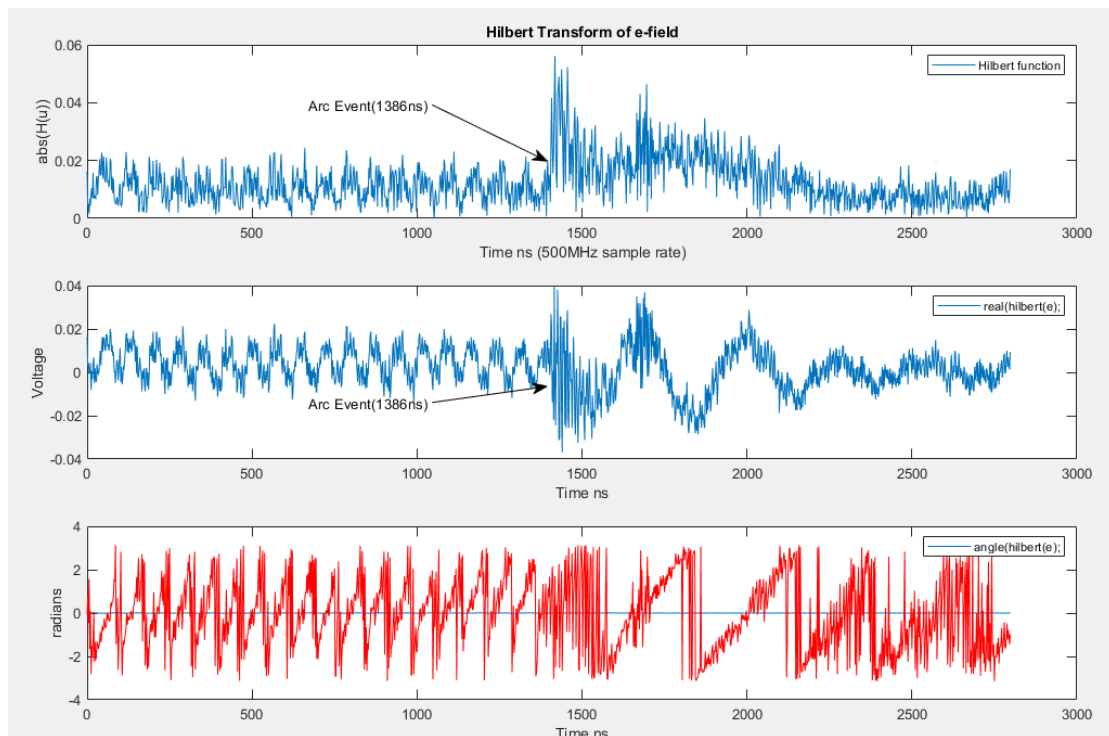


Figure 47 Hilbert Transform of E-field, Real Component of HT and Instantaneous Phase Angle of HT of the E-field

The absolute value Hilbert Transform of the E-field is not as distinct as that of the equivalent B-field, though it could be used as an event detector. A low pass filter on the HT instantaneous phase together with the creation of the low frequency electric field could be used on the E-field as a detector of events.

The 3D plot of the Hilbert Transform in Figure 48 and Figure 49 that follow indicates the steady state spiral of the fundamental drive frequency. The signal has some tight loops within it and these are due to the high frequency components in the measured signal. It is more usual to band-pass the acquired signal before applying the Hilbert Transform. Future research will determine the 'natural' frequency of the plasma, and bandpass this before applying the Hilbert Transform. This should clean the signal before being applied to the detector.

The 3D diagram is shown below and the steady state pattern is quite clear. Two observations are shown for clarity. The signal is the Hilbert transform of the data plotted with the amplitude as the Y axis and the instantaneous phase angle as the Z axis. If one plots just the real part as we have seen in the previous Figure (Figure 47) above, the signal graphed, is the reconstruction of the original signal.

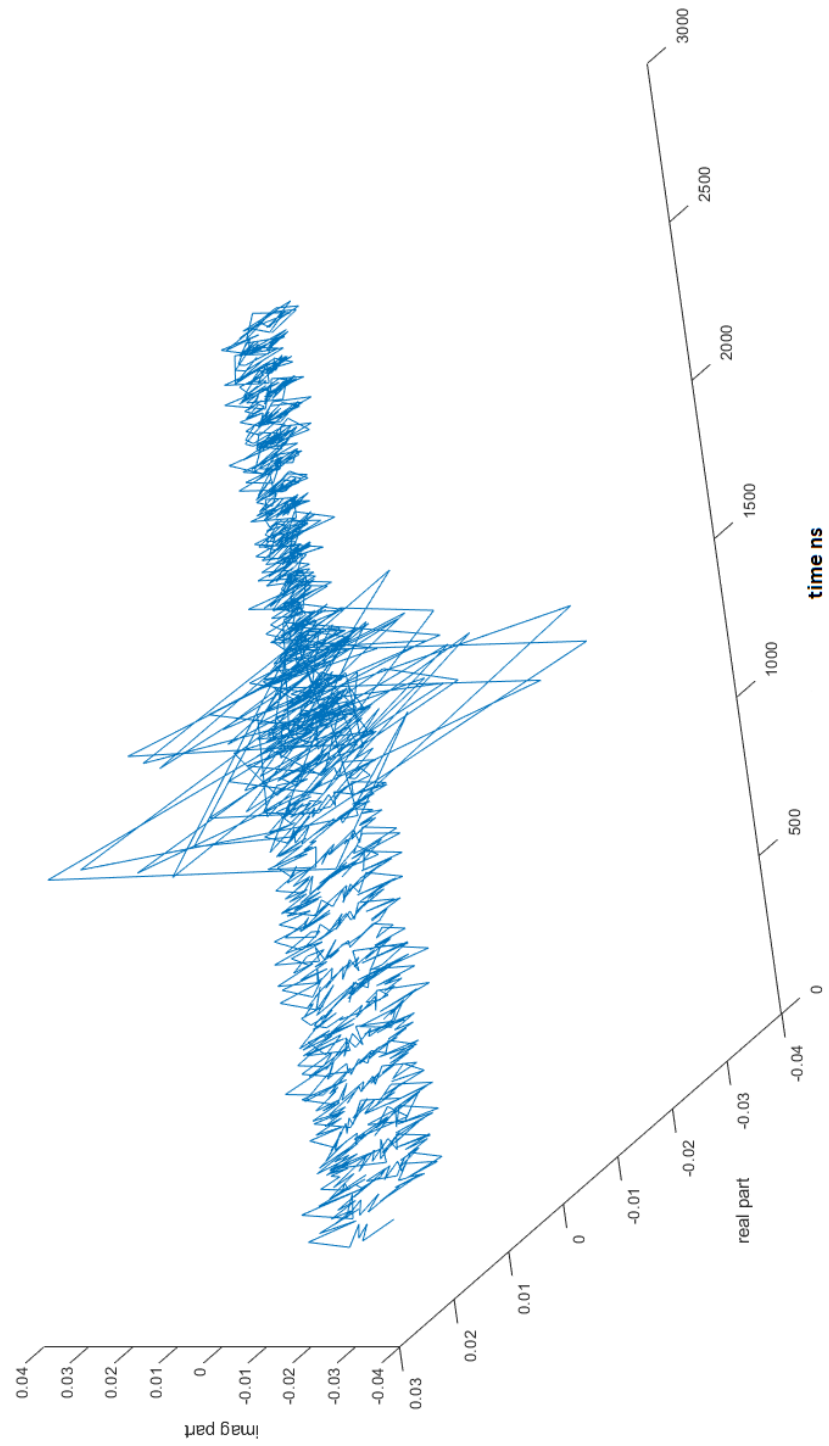


Figure 48 Hilbert 3D Plot Perspective #1, showing the Phase and Amplitude of the Hilbert Transform of the B-field.

We just plot the same data again but taken from a different perspective to show the relationship of phase to amplitude of the Hilbert Transform of the recorded B-field.

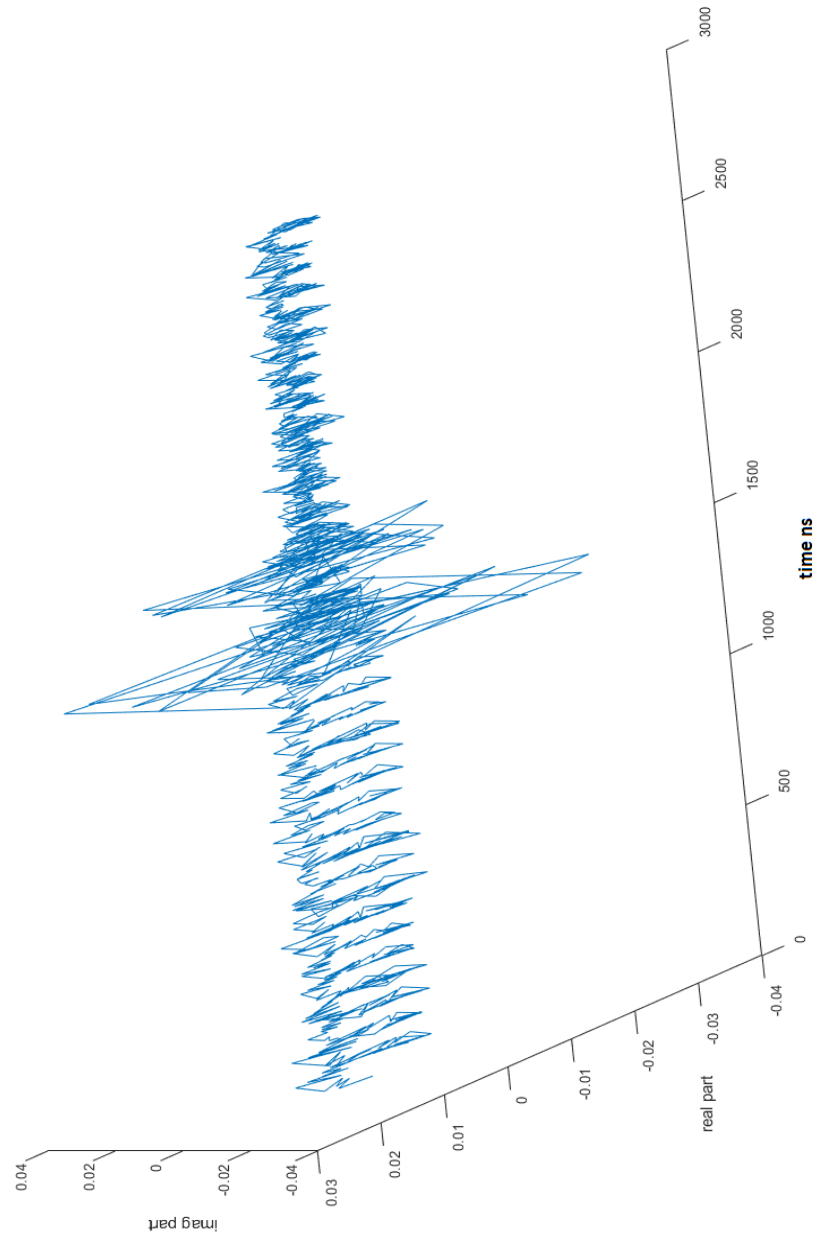


Figure 49 Hilbert 3D plot Perspective #2 showing the Phase and Amplitude of the Hilbert Transform of the B-field

We can see that the arc event is signified by a very high frequency burst in the B-field and that the instantaneous phase becomes a very high frequency signal in itself. At the 1350 ns location, there is a high amplitude B-field with a move from low frequency 13.56 MHz to some high frequency components. Then, the B-field amplitude decays, but still at a relatively high frequency. There is then a second high-amplitude B-field at 1700 ns and this comprises mainly high frequencies. By analyzing the HT data from the B-field, perturbations within the plasma can be detected and the phase/magnitude envelope for the current within the plasma in the time-frequency space can be monitored.

The E-field 3D plot for the Hilbert Transform of E-field follows the same form as above in Figure 48 and Figure 49, but with increased phase angle (spiral) around the arc event. It is possible that the low-frequency (below the RF drive frequency of 13.56 MHz) phase change could be utilized as a detector mechanism as the absence of the sub-drive indicates that the plasma is operating correctly. This requires further investigation.

4.7. Wavelet Transform Detection of Arcing Events

The Continuous Wavelet Transform (CWT) allows for the analysis of spectral data over time. It permits the investigation of frequency components over time and it is possible to perform time-localized filtering. The running system can be decomposed into a series of wavelets with coefficients for amplitude and time. The combination of these wavelets can be used to reconstruct the original signal.

4.7.1. Continuous Wavelet Transform Detection of Arcing Events

In normal operation without an arc event, we have seen that the Fourier Transform of the E- and B-fields shows in the main, the low-frequency components, i.e. the RF drive signal. The CWT shows these frequency components over time. The CWT of the B-field shows the main energy distribution in the RF drive frequency before the arc event and subsequent movement to the higher frequencies as the arc progresses. The analysis of the E-field with CWT shows a spreading of high-frequency components throughout the spectrum. (With the arc event, the movement from low-frequency to high-frequency is not as marked for the E-field as compared to the B-field. This will be shown later). If the arc events are included in the decomposition and tree creation for the CWT, then the production of high-frequency wavelets to reconstitute the arc event can be used as a detector as this indicates that the signal (or reconstituted version) contains high-frequency components indicating the occurrence of the arc event (Al-Shrouf, Saadawia, & Soffker, 2012). When the arc event occurs (at time base 1.35 μ s) we note the movement from the constant RF drive frequency of 13.56 MHz to the higher frequency components at time = 1.4 μ s and at time = 1.7 μ s of the B-field in the plot. This is clearly delineated in see Figure 50 and Figure 51 shown below.

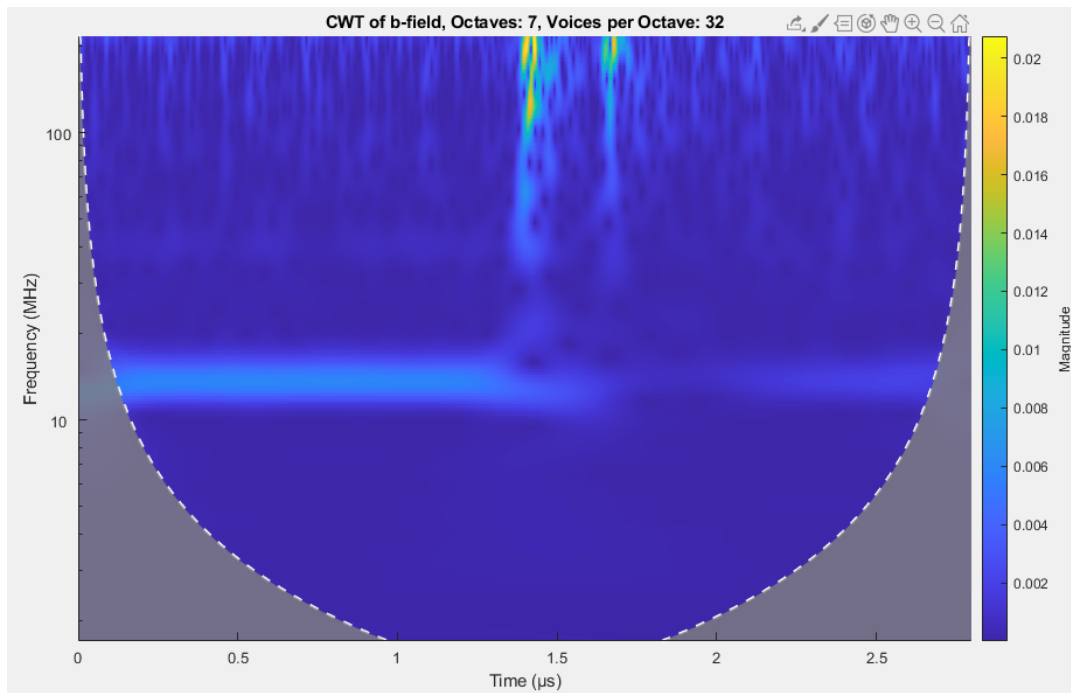


Figure 50 Plot of CWT of the B-field with Log Scale Frequency Bands with Confidence Curve

The following diagram is the plot of the equivalent CWT of the E-field.

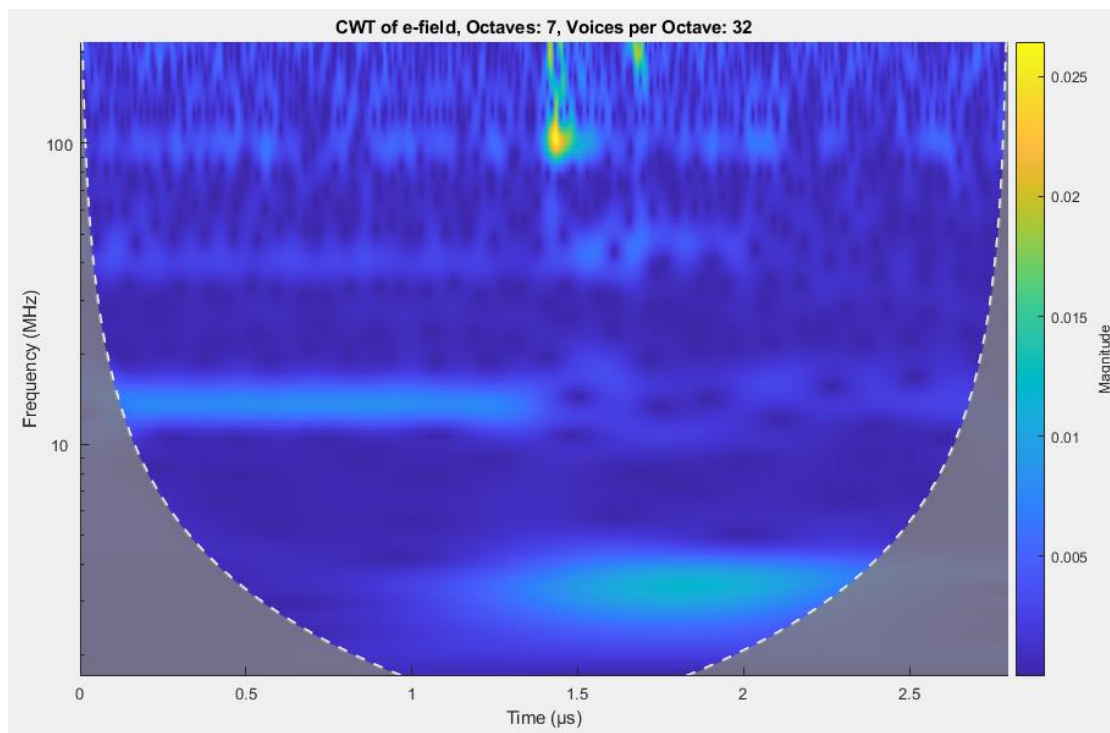


Figure 51 Plot of CWT of the E-field with Log Scale Frequency Bands with Confidence Curve

For the Matlab CWT to converge, certain constraints and limitations are set (in our calculations, we limited the number of voices to 32 and the number of octaves to 7 to cover the frequency span of interest i.e. 10 MHz to 150 MHz for the B-field analysis. We extended this for the E-field calculations to cover the low-frequency components post-arc.). Thus, the plot indicates a level of 'confidence' where these values can be calculated with a convergent solution.

Of note in the CWT of the E-field is the high energy density at 100 MHz, this not being a harmonic of the primary RF drive frequency. Also, of note in the spectrograms is the broad noise density throughout the e-field signal as compared to the B-field recording. The event is distinguished in the examination of the E-field by a brief power increase in 100 MHz range and production of a low-frequency component at 3.3 MHz, neither of which is a harmonic of the primary RF drive frequency. This requires further investigation.

For the B-field, the energy density moves to the higher frequencies as the arc event occurs at time 1350 ns.

4.7.2. Discrete Wavelet Transform for Detection of Arcing Events

The same technique was then applied using the Discrete Wavelet Transform (DWT). We want to examine this version of the transform as it is more readily applied to an FPGA real-time solution for deployment in the field. Using a set of cascade filters, the DWT can be calculated. Again, the original signal is as before and shown below is the block spectrogram of the signal in time, which is computed by the short-time Fourier Transform (in this case, the sample size is set to 128 and the default windowing is used). The Matlab toolbox short-term Fourier Transform is applied to the data set across the arc-event and the following plots are obtained.

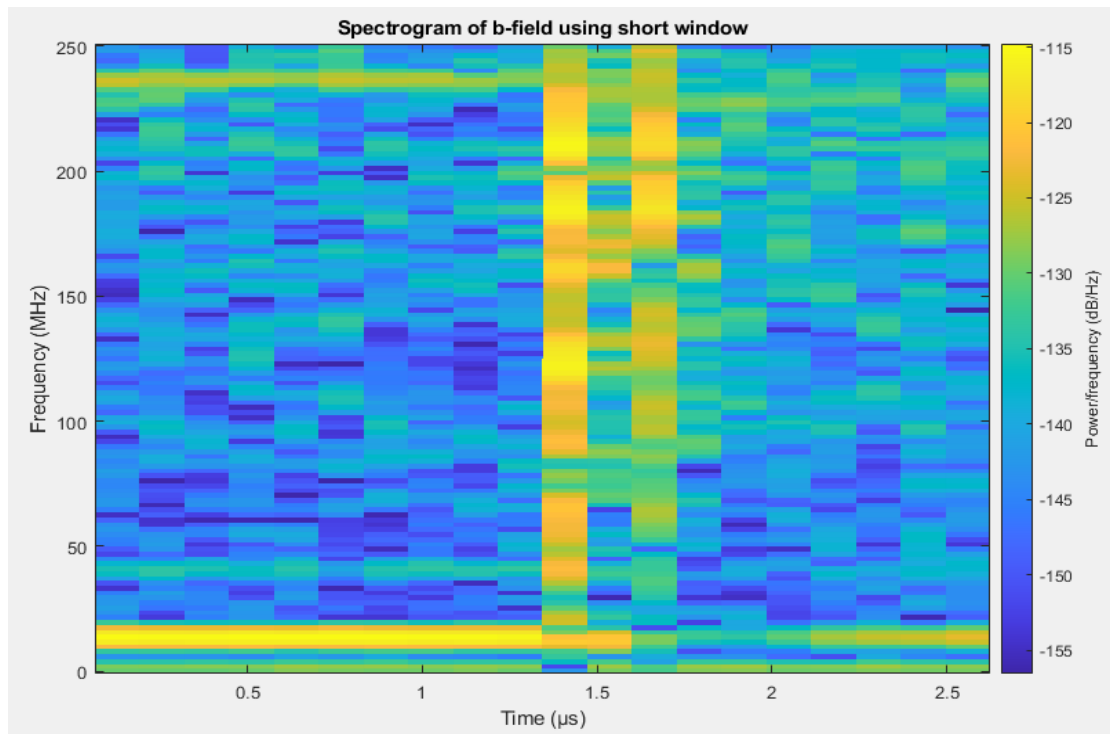


Figure 52 B-field Short Window Spectrogram Arc Event (at time 1.350 μ s)

The equivalent E-field spectrogram is shown below.

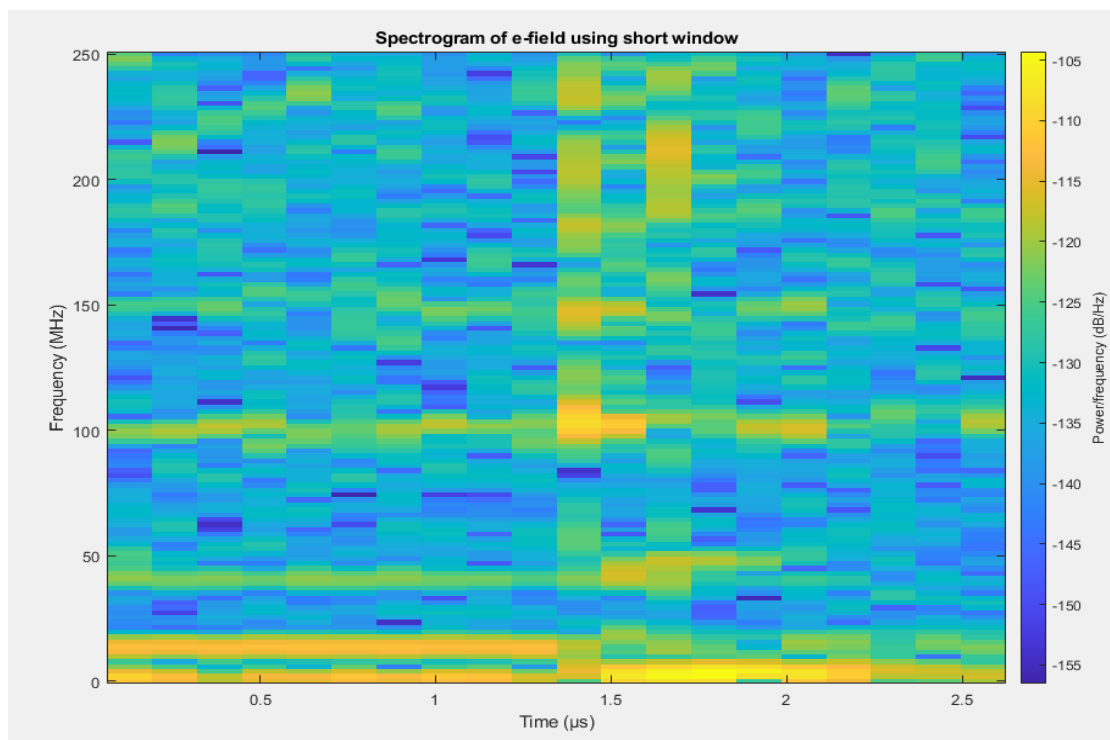


Figure 53 E-field Short Window Spectrogram Arc Event (at time 1.350 μ s)

Note that the scale on the Y-axis is a log scale of frequency in MHz for the CWT and linear one for the short-time Fourier transform. As per the CWT plots, the energy density is initially located in the 13.56 MHz up to the 1350 ns time. There is some spectral spread on this signal and we can see this in the original recorded data wherein the B-field does not appear as a pure sine wave. We have already seen that there are harmonics from the 13.56 MHz RF drive signal with the FRES system. When the arc event occurs, the power moves to the higher frequencies and we see the spectral density is greatest beyond 100 MHz. The graph shows that the power density of the 13.56 MHz before the arc event goes from -115 dB/Hz to -125 dB/Hz after the event. Frequencies above approximately 150 MHz for a pre-arc event go from -155 dB/Hz to -110 dB/Hz. The high-pass filters will detect this energy and the most significant output coefficients for the DWT will increase. We can see that by feeding these DWT coefficients into the KNN model, a classification can be obtained of the events as they are occurring in the operating plasma. This may be part of further work, i.e. to extract the DWT coefficients and build a KNN or RBF model to detect the different kinds of arcing events and properly classify them.

4.7.3. DWT Cascaded Filter Design

As can be seen from the B- and E-field analysis, in Figure 52 and Figure 53 above, the Discrete Wavelet Transform (DWT) is composed of a series of cascaded special low-pass (LP) and high-pass (HP) filters with decimation at each stage to meet the Nyquist criteria. The filters operate in parallel and cascade into pairs to divide the spectrum high and low. This process is repeated until the signal processing

spectrum is divided into the granularity required for the coverage of the expected signal spectrum. Note that the Matlab software implementation has some limitations and constraints in order to converge to a result in a suitable time-frame. This set of cascaded filters is shown in Figure 54 below, as is the data flow in the DWT. The high-speed data is acquired by the analogue front end, is digitized and then passes through the series of filters. As can be seen, this is a cascade of digital HP and LP filters with suitable decimation for the frequency at each stage. (For the lower frequency filters, decimation involves just dropping samples). This can be readily integrated into the real time acquisition of the E- and B-field in a FPGA implementation with digital filters and decimation units. The output coefficients of the filters can be applied to the K-nearest neighbor (KNN) and radial basis function (RBF) neural network system described in the next chapter to classify the detected signals.

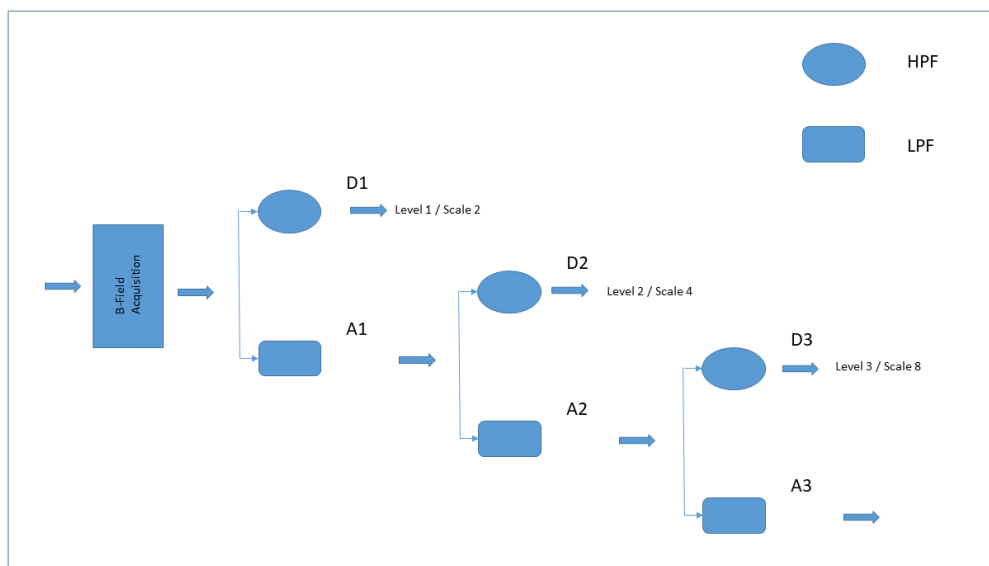


Figure 54 Discrete Wavelet Transform Data Flow using HPF and LPF

The base scale of this is based upon the binary scale. This gives dyadic scaling and shifting

$$\text{Scaling} = 2^j \ (j = 1, 2, 3, 4 \dots)$$

and the translation is performed in multiples of this:

$$\text{Translation} = 2^j m \ (m = 1, 2, 3, 4 \dots)$$

The function compares the incoming signal to discrete multi-rate filter banks and the outputs of the filter are the coefficients of the Discrete Wavelet Transform (DWT).

4.7.4. Reconstruction of Signal from Wavelet Transform

Using the Wavelet Transform and its coefficients (before applying any compression, Hamming code or decimation), the signal can be reconstructed. We apply the Inverse CWT (ICWT) to generate this 'reconstructed signal'. We can use this reconstructed signal to test the efficiency and accuracy of the Wavelet Transform of the input signal. Note that for an FPGA implementation for a practical detector, the DWT would be used. The reconstructed signals are shown in the figures (Figure 55, Figure 56, and Figure 57) below, with plots of the original signals as recorded by the Rigol and the reconstituted signal from the ICWT. It can be seen that the B-field can be accurately modelled with components from just below the RF drive frequency while the E-field requires frequency components from well below the primary RF-drive frequency. For the ICWT, the accurate

reconstruction of the B-field is from 10 MHz to 150 MHz and for the equivalent E-field the range is from 2 MHz to 150 MHz. This is to cover the low-frequency component that is generated in the E-field when the arc event occurs. In both cases, the reconstructed signal matches very well, though for the same constraints in octaves and voices, the B-field is constructed more accurately.

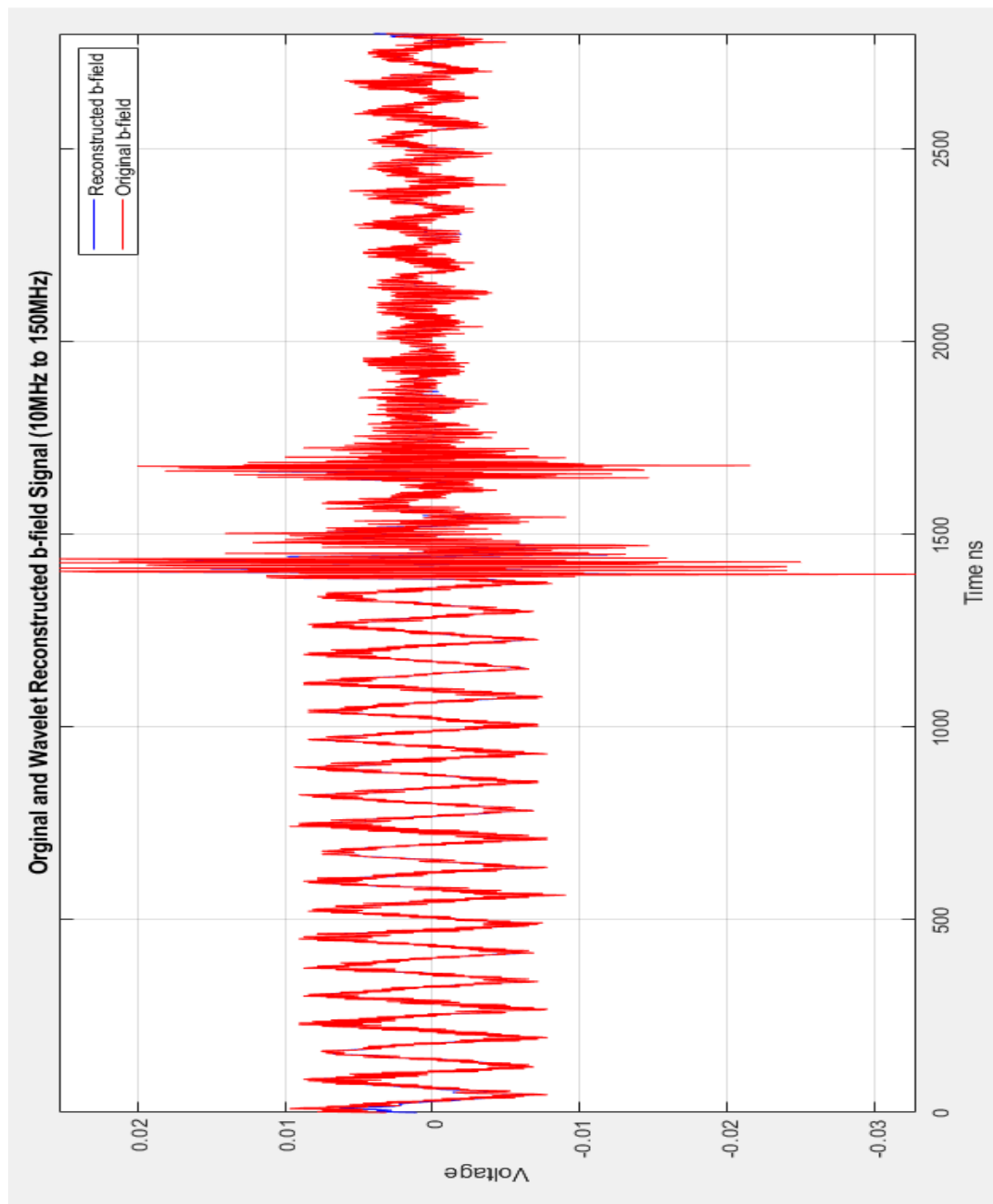


Figure 55 Reconstructed and Original B-field Signal from Inverse CWT (almost indistinguishable)

These signals are almost indistinguishable and the differences are only noticeable when zoomed in.

With the Matlab® toolbox, the number of octaves and the number of voices per octave of the frequency span for CWT is the limiting factor. For our examination, we tested with both fine grained and limited frequency span and with course-grained and wider frequency span. We see from the CWT that the energy moves to frequencies greater than the reconstructed frequency of 150 MHz. We can perform the CWT and the inverse CWT using a coarser granularity. Therefore, it is only the frequencies greater than 150 MHz that are observed to show the discrepancy of the inverse CWT. For the B-field, this is shown below.

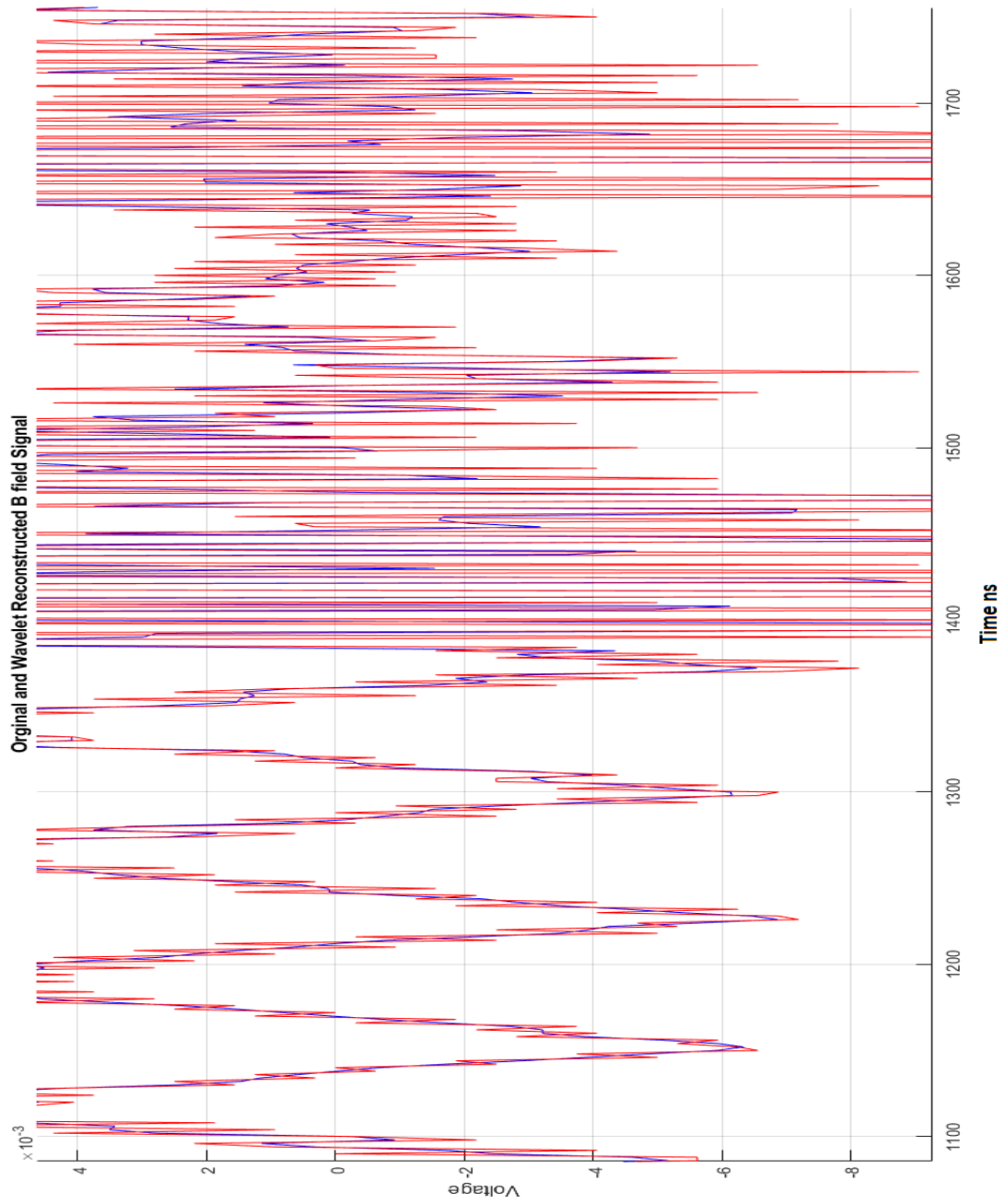


Figure 56 Inverse CWT B-Field Showing the Reconstruction Deviation at Higher Frequencies

The corresponding reconstructed E-field signal is shown in Figure 57 below:

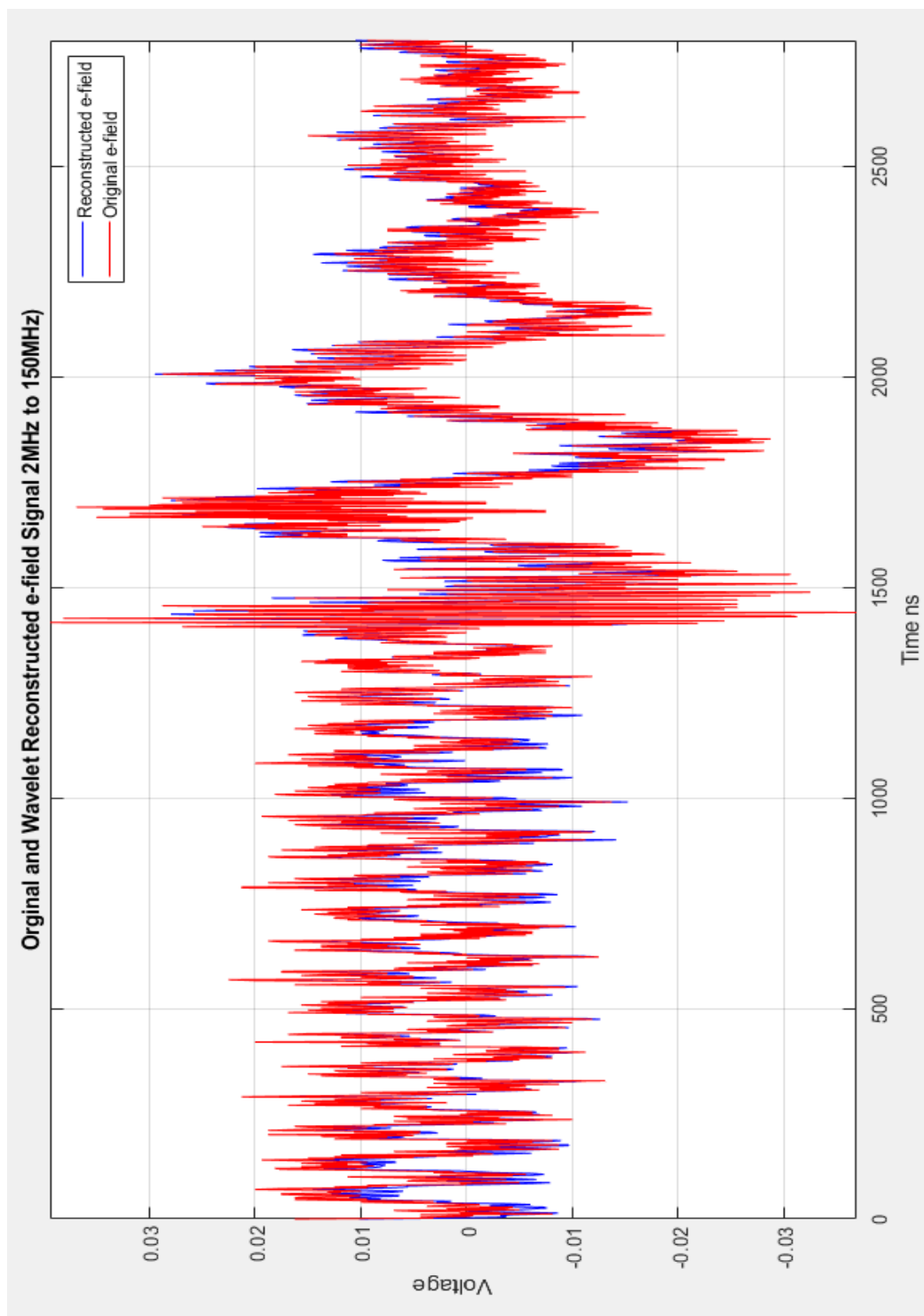


Figure 57 Reconstructed and Original E-Field Signal from Inverse CWT

The reconstructed E-field is readily discerned from the original signal with its high-frequency variation.

Slight variation in the reconstructed ICWT signal can be seen, but this should not affect the basic detection technique of a DWT HF filter output for detecting arc events. The variation can manifest itself via the absence of higher frequency components in the reconstructed signal.

4.8. Plasma Arc Event Detection Techniques

For the Oxford Instruments PlasmaLab 100 CCP tool, these three demodulators can be used to detect arcing events. The outputs of these detectors may be fed to neural net classifiers to determine the type of event that is occurring within the chamber. The initial decoding has been focused on the Oxford Instruments PlasmaLab 100 CCP tool with a primary RF drive frequency of 13.56 MHz. There is future work to narrow-bandpass the natural frequency of the plasma and apply the above techniques. The B-field for the TRES system for equivalent antenna locations produces a less noisy signal, and may be more suitable for these event detection systems. With the DWT filtering system, the vectors could be applied to an RBF or KNN subsystem and the operating events of the system could be classified. These novel plasma parameter detection techniques very much lend themselves to the characterization of chambers from an operations point of view, e.g. tool matching and also to the execution of recipes on a machine in an industrial environment. With the rich raw data of the TRES system and these computer digital-signal-processing (DSP) techniques combined with powerful AI techniques, complex but accurate systems may be developed for monitoring and controlling industrial plasmas in a production environment.

4.9. Summary

The TRES processing techniques described are of great benefit to plasma processing operators as they may be used in real-time control mechanisms and also historical data processing. By combining the advantages of the real-time acquisition and the digital signal processing techniques, powerful tool-side monitoring equipment can be deployed. This can lead to enhanced quality, lower cost and improved performance of the complex plasma processing steps in an industrial environment. Accurate event-detection within a plasma and even real-time monitoring of its correct recipe performance, are vital for the continued improvement in plasma processing. With the simple arc-detection schemes to the continuous vectorized AI processing of the recipes, these systems are greatly advanced over the simple parametric measurements that are made tool-side today.

This page intentionally blank

Chapter 5 Simulation Models of Capacitively Coupled Plasma

5.1. Introduction

We have seen the basic example of the equivalent circuit of the capacitively coupled plasma in Section 2.1. We further extend this model introducing enhanced models to map the recorded harmonic content of the Oxford Instruments PlasmaLab 100 CCP machine. There are two main motivations for this, one being the real-time calculation of plasma properties as the plasma is operating and the other is to permit the closing of feedback loops on the control of an industrial machine and the process. For example, in reactive sputtering, *“the accurate prediction of thin film properties cannot be measured in situ or during the deposition process”* (C. Woelfel, 2019 (Oct 8)). The motivation is that by using the radio-spectra measurements and mapping those to a conveniently calculated electrical circuit model, the adjustments for the various process parameters may be targeted to quantifiable circuit components. These may be adjusted to and verified for a process control feedback mechanism. With reproducible steady-state process conditions and radio-spectra measurements, investigations of plasma control feedback become possible.

Further investigations beyond the scope of this work include the modulation of the primary drive RF to see if the damping of the plasma for the CCP with a single source is a non-linear damped system and to extract values for the damping factor μ . The type of system emerges from, for example, models of a limit-cycle or Van der Pol oscillator that requires an RF drive. This limit-cycle oscillation may be represented from a mathematical point of view by

$$y'' + \mu(y^2 - 1)y' + y = b$$

where

b is the forcing function (in this case the RF drive at 13.56MHz)

μ represents the nonlinearity and strength of the damping.

When $\mu = 0$, the equation becomes $y'' + y = 0$, which is a simple harmonic oscillation around the RF drive frequency. For $\mu > 0$, the system enters a limit-cycle; near the origin ($y = y' = 0$), the system is unstable; far from the origin, the system is damped. This can be seen in the diagram below (see Figure 58).

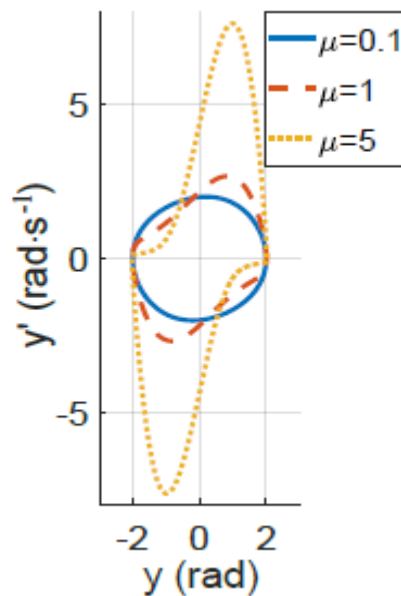
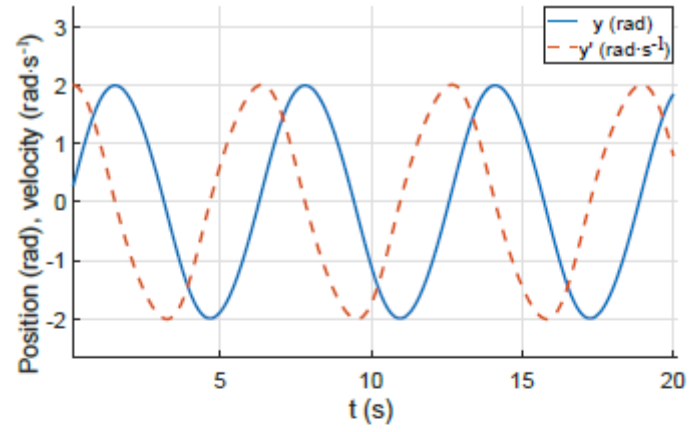
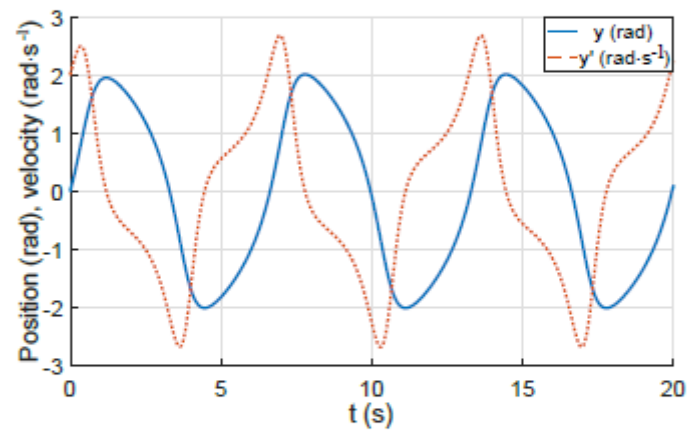


Figure 58 Limit cycles of unforced Van der Pol oscillator representing a limit cycle system

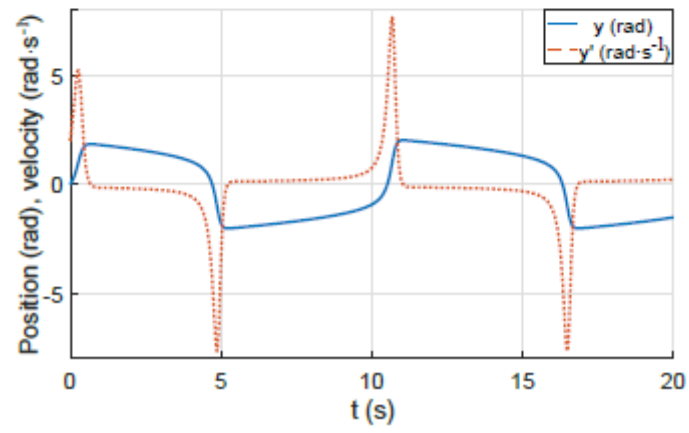
The resulting modification of the driving RF sinusoid can be calculated as below for the different values of μ for the time domain signal is shown in the figure below (see Figure 59) (Vaclav Satek, 2019 (Sept)).



(a) $\mu = 0.1$.



(b) $\mu = 1$.



(c) $\mu = 5$.

Figure 59 Van der Pol oscillator in time domain with varying values of μ (Vaclav Satek, 2019 (Sept))

This work is outside the scope of this thesis, but further studies may adapt the calculation of the Taylor series expansion for the time-domain output of the Van der Pol oscillator to the time domain signal as measured and shown in Chapter 4.

We will initially examine the basic electrical model as proposed by Schmidt *et al.* (Frederik Schmidt T. M., 2018). We expand upon the basic circuits to closely match the observed radio-spectra and derive circuits that can be simulated in readily available circuit simulators. This permits rapid deployment in an industrial environment with a robust calculation of model values for the observed radio-spectra and the determination of the operating process state of the plasma system.

5.2. Non-Linear Plasma Modelling

As we have seen, the basic electrical model for the plasma is a lumped network of linear components, one of the most comprehensive examples which is shown below (Figure 60):

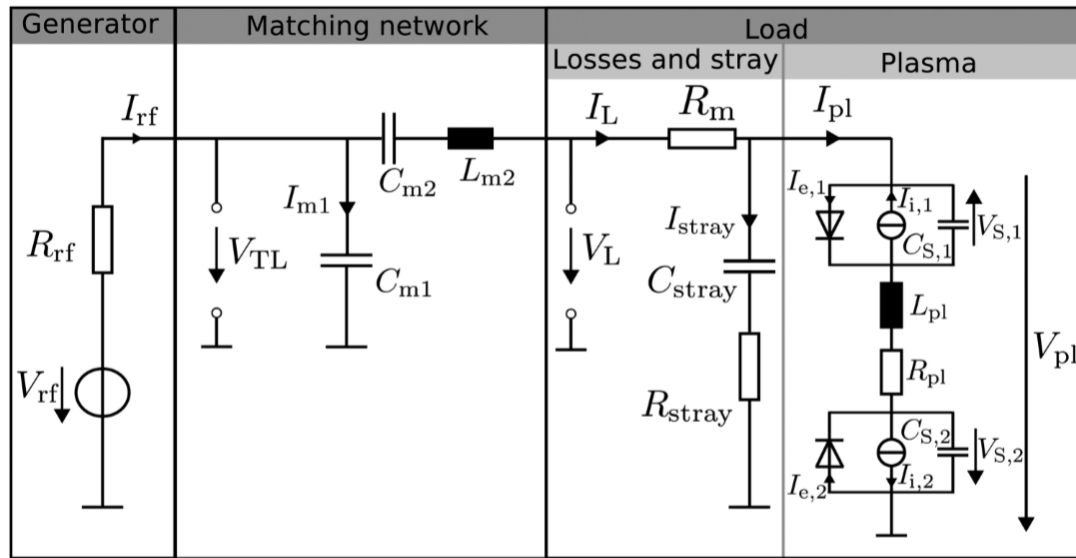


Figure 60 Generator, Matching Unit, Cabling and Non-Linear Plasma Equivalent Circuit (Frederik Schmidt T. M., 2018)

The diode is introduced to the sheath circuit to model the non-linear effect and the bulk is modeled as a resistive/inductive element. For simulation, the diode model has three linear regions of conduction: on, off and reverse breakdown, though the breakdown mode is not utilized. In Schmidt, (Frederik Schmidt J. T., 2019), we see

the simulation of the circuit using a linear circuit connected to a non-linear plasma model. The transfer matrix describing this non-linear (plasma) component is calculated using an iterative method evaluated over the transient currents entering the plasma as a finite Fourier series by evaluating over a small applied voltage (of the order of $1/100^{\text{th}}$ of the main drive) for the harmonic frequency of interest. The model developed was for a symmetric chamber. For this simulation of the symmetric controlled chamber, they report very good simulation results compared with those observed, captured using a voltage-current (VI) probe on the generator cable for the first 12 harmonics.

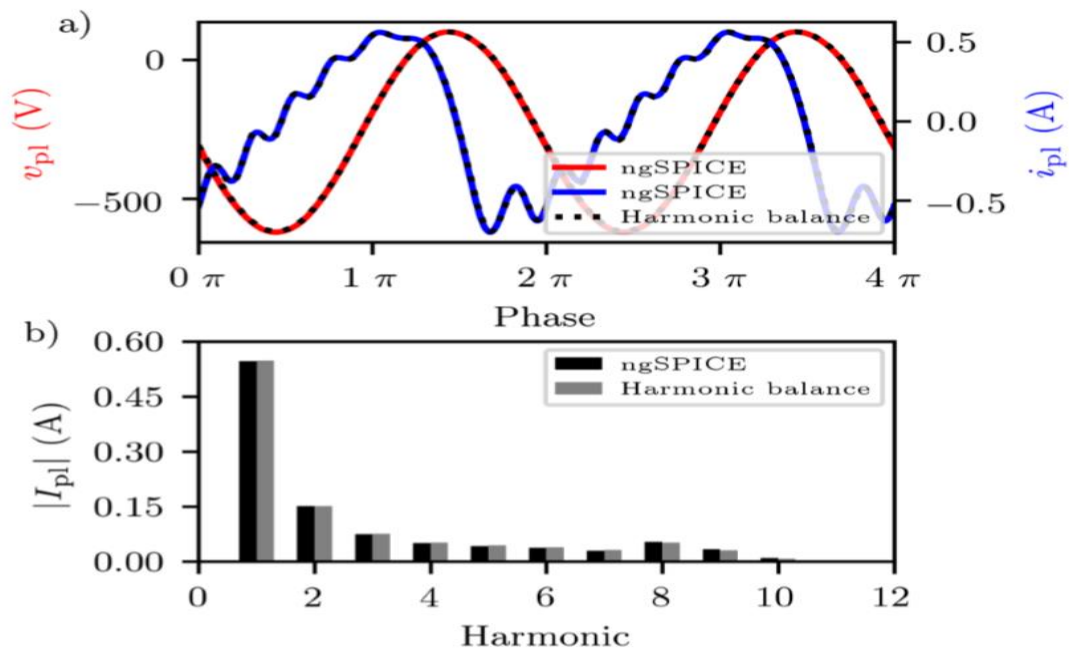


Figure 61 Voltage and Current Using the Global Plasma Model (Frederik Schmidt J. T., 2019)

(a) Transient solution. The solid lines indicate the results obtained using ngSPICE, while the dotted lines represent the results obtained using the harmonic balance algorithm. (b) The absolute values of the different current harmonics. In black are the results obtained by ngSPICE, and in gray are those that are obtained from the harmonic balance algorithm. (Frederik Schmidt J. T., 2019)

The Oxford Instruments PlasmaLab 100 has a high order of asymmetry (as can be seen in the photo (Figure 62) below with a central active plate and the large cavity area connected to ground). Also, the monitoring system in use (Kelly & McNally, 2017) is placed where the ports are available on the machine giving rise to additional asymmetries of the radio spectra B field measurement.

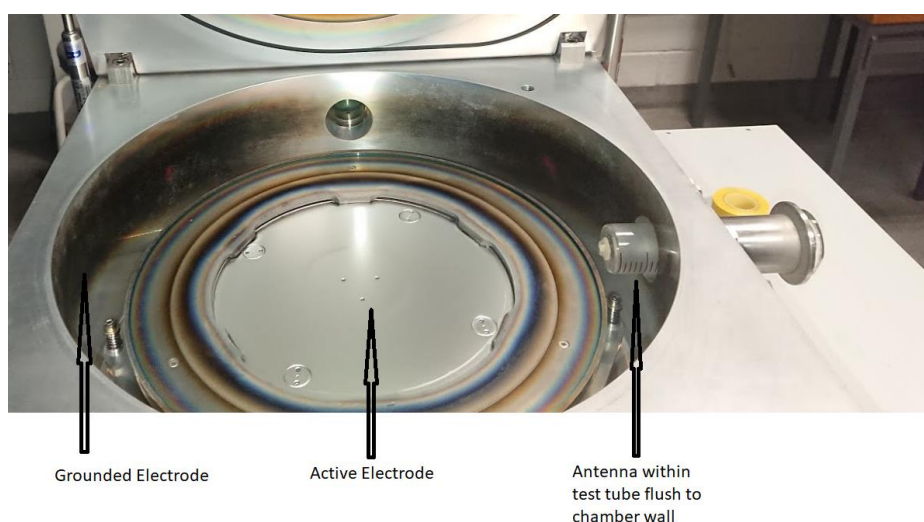


Figure 62 Asymmetric Electrode Plasma Chamber with Location of Flush Antenna.

As can be seen from the photo (Figure 62), the antenna is radial to the plasma and axial to the bulk current flow. As the voltage applied to the drive plate increases, the amplitude of the fundamental frequency of the discharge current increases, but this is not so for the higher harmonic frequencies (Dvorak, 2013). A difficulty with the B-field antenna system is that it does not take account of the specific areas of the high frequency harmonic generation location. This spatio-temporal data is lost due to a number of factors:

- Axial height between the plates
- Radial location from the axis of plasma

- View of both sheaths which are asymmetric
- Near field $\frac{1}{r^3}$ falloff
- Dynamic range of antenna (E-field rejection) with respect to harmonic signal levels
- Frequency response / transfer function of the RF antenna
- Averaging function of spectrum analysis recording.
- Skin effect within the sheath boundary, which effects different frequencies to differing degrees.

For the Oxford Instruments PlasmaLab 100, up to 102 harmonics were discernible and recorded for the experiments with the antenna and spectrometer configuration as shown in Chapter 3. In later chapters we will describe examinations and transform methods applied to the harmonic data to determine the operating parameters of the plasma. Earlier examinations were with the first few harmonics (Frederik Schmidt T. M., 2018), but as can be seen in Chapter 4 and later chapters, the various higher order harmonics are very informative of the operation of plasma processes. The recent work of Dvorak *et al.* uses the static harmonic-balance matrix approach in a symmetrical chamber for a particular operating mode (Dvorak, 2013). All of these matrices would have to be calculated for each of the steps of the process for a particular recipe within a plasma chamber. With the chamber running a known portion of the recipe, the correct matrix would be then selected. This matrix would then be inserted into the global model to deduce the operating parameters of the 'steady-state' plasma. For a multi-frequency chamber with hundreds of harmonics of each frequency and possibly

beat-harmonics being recorded, the non-linear harmonic-balance matrices could become quite unwieldy.

Lieberman *et al.* recently conducted an analysis of a symmetric large-area capacitively coupled discharge (where the wafers being processed correspond to half a metre in area) driven by high frequencies up to 200 MHz (M A Lieberman, 2002 (June 14)). They concluded that the observed inhomogeneities are caused by electromagnetic effects which were classified as standing wave and skin effects using a transmission line model.

Analytical and numerical methods have been employed to investigate the operation of plasma systems (R.P. Brinkmann, 1996). Such numerical methods include Monte Carlo and particle in cell (PIC) techniques (R.W. Boswell, 1987) (Kushner, 1988), fluid-dynamic calculations (E. Gogolides, 1992) (Sk Park, 1990) and kinetic methods (H.Leyh, 1998). These models give quantitative values for the external control parameters of the plasma and the various internal quantities. *"In spite of this effort, however, it seems that even today most progress has been in the design and the characterization of plasma processes is achieved by purely empirical methods"* (R.P. Brinkmann, 1996). In the electrical circuit model, we lose much of the spatial-temporal framework, but rapid calculation permits an industrial go/no-go processing control system.

The equivalent circuit models using lumped components will model the primary sinusoidal current but are far less effective in simulating the observed plasma RF harmonics. In low pressure CCPs, while the RF energy is being applied, *the electron power gain occurs at the sheath expansion* (Sebastian Wilczek, 2016). This is due

to the electric field reversal during sheath collapse. While there is sheath expansion, beams of energetic electrons are generated, which penetrate into the plasma bulk. (This temporal phenomenon will be examined in further work). These beams of energetic electrons have a unique rise and sustain time during a portion of the RF cycle and sustain the discharge through ionization. Using Particle in Cell / Monte-Carlo collisions, (PIC/MCC) simulations, Wilczek *et al.* (Sebastian Wilczek, 2016) were able to calculate electron density and movement in a spatial dimension at the electrodes over the application of an RF cycle. This produced the following density map for an Ar plasma with an applied cycle of 55 MHz (18 ns cycle time).

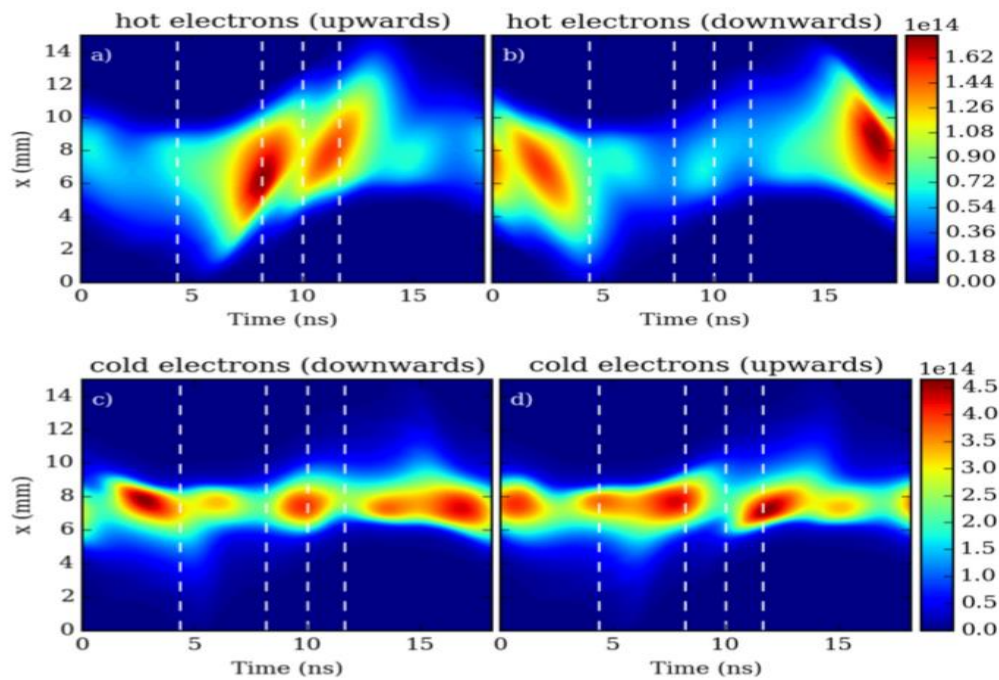


Figure 63 Hot Energetic Beams of Electrons During RF Cycle (Sebastian Wilczek, 2016)

In these spatio-temporal plots of the density of 'hot' electrons ($\epsilon > 11$ eV) that move upwards(a) and downwards(b) as well as the density of 'cold' electrons below 4 eV that move downwards(c) and upwards(d) are calculated. The time axis covers one RF period (~18 ns). The densities are given in m^{-3} . The powered electrode is located at $x=0$ mm while the grounded electrode is located at $x=15$ mm.

The penetration of these beams of energetic electrons into the plasma bulk starts as the sheath begins expanding due the applied potential. The PIC model was for a symmetric electrode, so equal beams occur at the grounded electrode half an RF cycle later. For the Oxford Instruments PlasmaLab100, the asymmetry would affect the timing and density of the beams. As the sheath expands, the beams of electrons gain high velocities. This leaves a space of positive charge behind which attracts colder bulk electrons towards the powered electrode which have an inertia. This leads to powerful non-sinusoidal current effects between the bulk and the powered electrode.

It can be seen that the non-linear action of the plasma is quite difficult to model given the variable parameters of the measurements, the plasma constitution and the enclosing chamber and electrodes. We examine a scalable electrical model to match the observed measurements so as to build descriptive operating point for the plasma process in a production environment to close the loop on its correct operation. The extended model can be calculated in real-time, and evaluated for correct operation for each part of the recipe of a plasma process and appropriate actions can be taken should the plasma operation be determined to be out of bounds.

5.3. Capture of B-field Harmonics in a Capacitively Coupled Plasma System

The B-field was recorded using the custom RF ID plain-coil loop antenna and the SignalHound SA44B Spectrum Analyzer. By recording as many harmonics as was practical, the multi-node extended electrical model could be tuned to match the

recordings. With the antenna located outside of the chamber, the higher harmonics were very attenuated (6th was reduced to -100 dBm). (Note: The wavelength of the primary drive is ~22 metres and the 35 mm KF40 opening may appear as a 'slit', i.e. $\lambda_{harmonic} \gg \lambda_{slit}$). To obtain a better signal, the following configuration was employed. The B-field antenna was inserted into the test-tube until it was flush with the inside wall of the chamber of the operating plasma. An Argon plasma was operated with fixed parameters to get a level steady state and the frequency sweep was acquired with the novel patent-pending cabled balun system ["Electromagnetic field signal acquisition system for high signal-to-noise ratios, and electrical noise immunity", GB Patent Application No. GB2005828.5. - filed 21 April 2020.] and the SA44B spectrum analyzer.

Using the photo in Figure 62 above as to the layout of the electrodes and the relative location of the KF40 port, the following picture (Figure 64) shows a close-up of the antenna location in the plasma chamber wall. The antenna is inserted into the test tube until it is flush with the chamber wall. It can be seen that the antenna is co-located between the electrodes and radial to the bulk plasma.

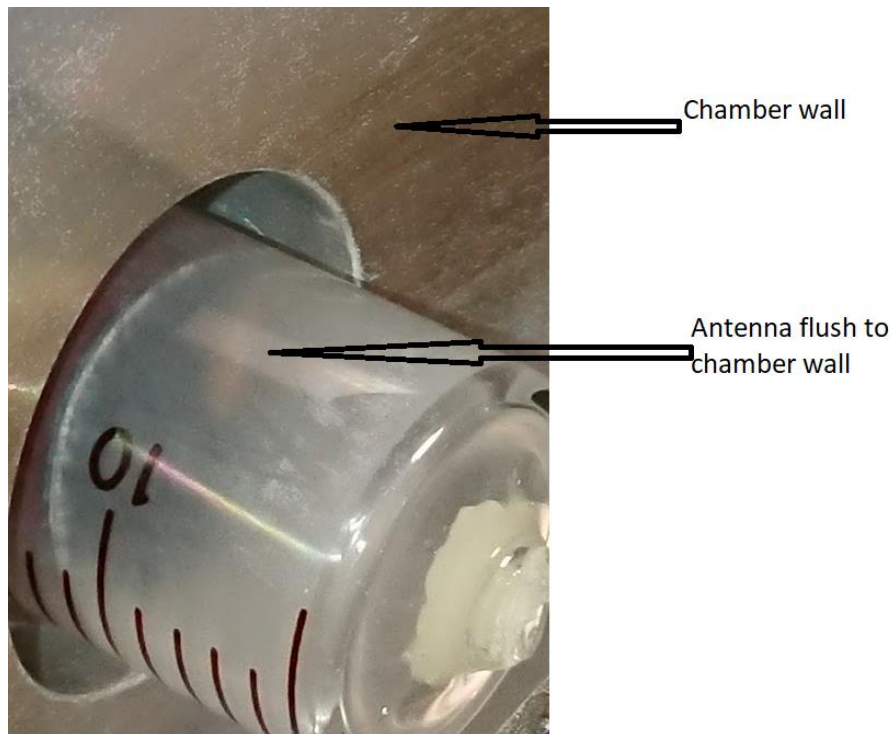


Figure 64 Antenna Flush to Oxford Instruments PlasmaLab 100 CC Plasma Chamber Wall

The following (Figure 65) is a plot of the measured near-field B-field harmonic content. The initial high-power RF drive fundamental is excluded from the acquisition such that the reference level of the spectrum analyzer is not overloaded. The picture below (Figure 65) is an overlay of the background signal (the low underlying grey plot) and the harmonic reception with the Ar plasma operating. This is to compare the reception of (higher order) harmonics with the background noise. The active harmonics can be clearly observed over the underlying noise. The high order harmonic signal level from the plasma is of the order of 10 to 20 dB greater than the background noise. When evaluating changes in the values of the harmonics, values of the order of 0.25 dB are significant. When taking background measurements, it is found that the active spectra region in the 850 MHz area is being received from an adjacent mobile phone cell tower. The harmonic content from the plasma in this region is relatively low. When the

antenna is withdrawn from the test-tube and operates outside the wall of the chamber the spectrum of the high-frequency components is severely attenuated such that approximately only the first 10 harmonics could be ascertained from the underlying noise floor.

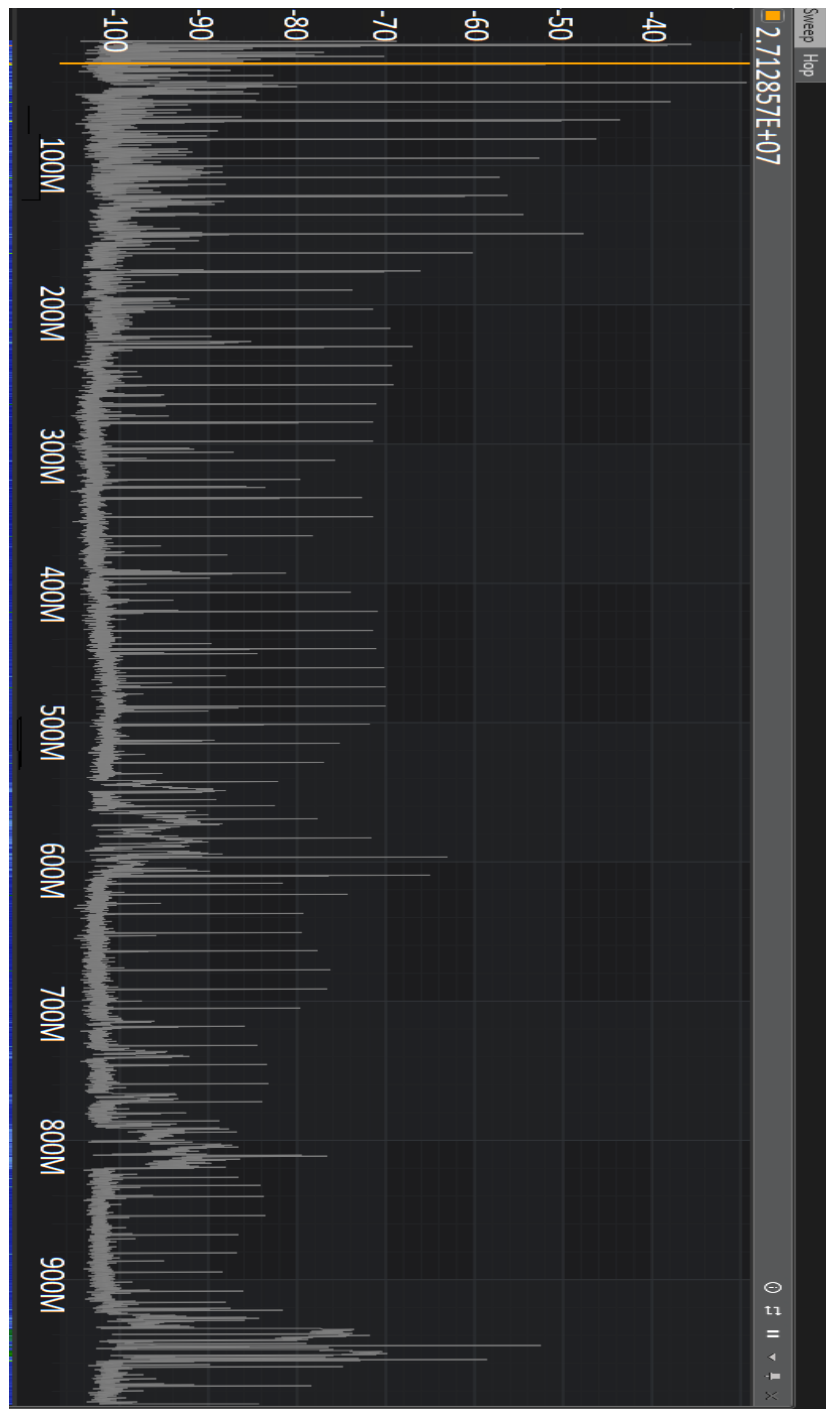


Figure 65 Harmonic B-Field Spectrum 20 MHz to 1 GHz (power scale is dBm)

To extend the operation of the equivalent electric circuit (Lieberman M. A., 1988) for the asymmetric electrodes and the variance in capacitance at the electrodes due to the applied field, the equivalent circuit for the plasma is extended with varactors in place of the diodes. The varactors combined with the biasing network permits the tuning of the capacitance variation due to the applied field for both the powered and the grounded electrode. We also extend the internal tuned components to account for the high-frequency resonances as seen in the spectrum. This permits the electrical modelling of an equivalent circuit that matches the real-time acquisition of harmonic content and to determine the typical “go/no-go” operation of a plasma system within an industrial environment. In the later chapters, by examining the principal component analysis (PCA) of the harmonic content, the high-order harmonics relay significant information on the operation of the plasma. It will also be shown that significant information can be gleaned from the higher-frequency components for detection and analysis for transient events within the plasma (arcing) (see previous chapter).

5.4. Revised Equivalent Circuit and Spice Simulation.

The revised circuit is as shown below (see Figure 66). We note that the plasma operates in a non-linear mode insofar that by applying a single-tone sine wave, we record a complex current and voltage waveform at the viewport and this produces a high order harmonic waveform when viewed in the frequency domain. What is also of note is that all the harmonics are present to some extent, not just the even or the odd ones. This reflects the observed experimental situation and is in contrast to previous modelling: *“Even harmonics are not excited due to the symmetry of the discharge”* (Sebastian Wilczek, 2016) indicating an asymmetric

discharge. The equivalent diodes in the circuit perform as mixers which implement this frequency multiplying effect. In the model we initially introduce a diode whose capacitance is a function of the applied voltage as per the voltage applied to the sheath. For consistency and comparative purposes, we model the matching networks necessary for operating capacitively coupled plasmas from the paper by Schmidt, Mussenbrock and Triesmann (Frederik Schmidt T. M., 2018). We have omitted the steady-state reverse ion current sources as we are interested in the dynamic simulation and the corresponding FFT of the time-domain response. We will examine later, incorporating these hot and cold electron beams as phase-controlled sources to the applied RF to better match the “hot and cold” electrons that are injected into the sheath as shown in Figure 63 above. This gives us the equivalent circuit as shown in the diagram below (Figure 66).

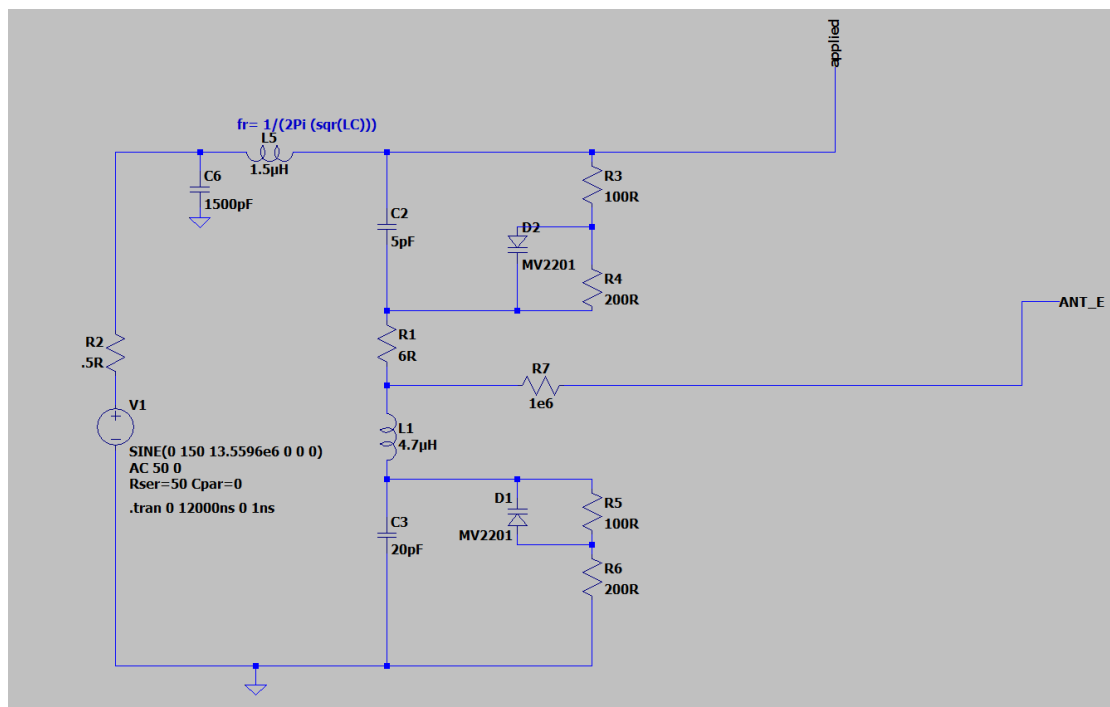


Figure 66 Capacitively Coupled Plasma Equivalent Circuit Model with Varactor

The potential divider around the diodes is simply to adjust the applied voltage to the varactor to adjust its capacitance range with respect to the fixed capacitor representing the standing sheath capacitance. The MV2201 varactor has capacitance range of approximately 15 pF ($0.1 V_r$) to 3 pF ($30 V_r$). When the applied voltage from the generator is set at 150 V at 13.56 MHz, the following time-series result is obtained for the voltage within the simulated plasma bulk.

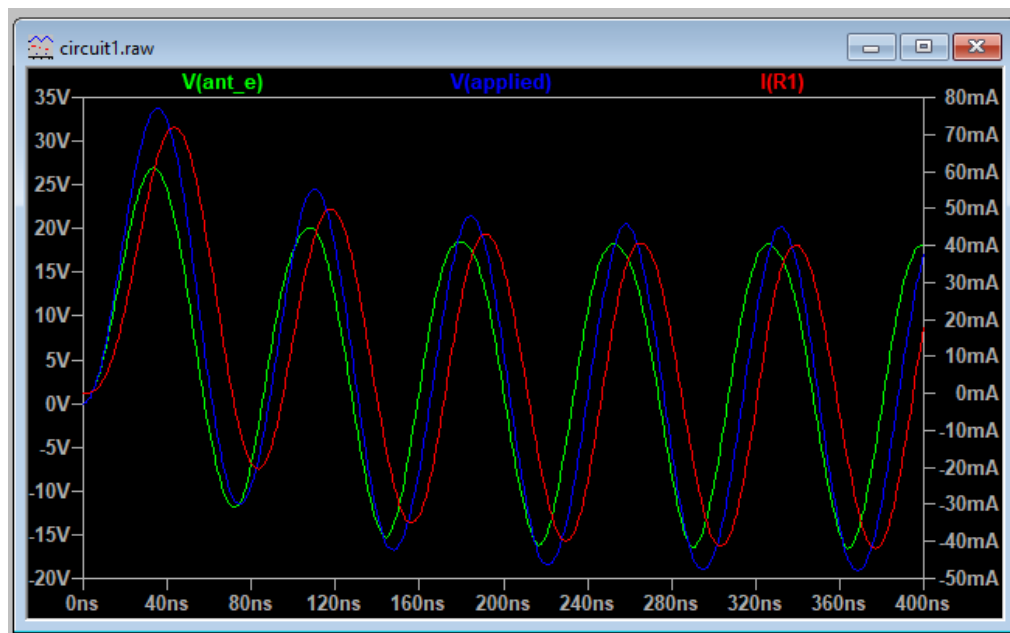


Figure 67 13.559 MHz Simulated Equivalent Circuit with Tunable Varactor CCP Time-Series

In the time series simulation, the output is direct coupled and includes a charging effect from the two diodes with the non-symmetrical capacitors. This leads to a dc-charge pump of the bulk. The measurements taken with the B-field antennae are inductively coupled and those of the E-field are capacitively coupled and no bias will be recorded. The V_{ant} is the monitored voltage at the centre of the bulk. The measurement of $V_{applied}$ is a monitor of the applied voltage after the simulated matching unit. The bulk current is monitored as the current through the resistor

R1. This is shown as the red plot in Figure 67 above. The output is slightly non-sinusoidal and when the FFT is taken of this output voltage, the high order harmonics are observed (Figure 68).

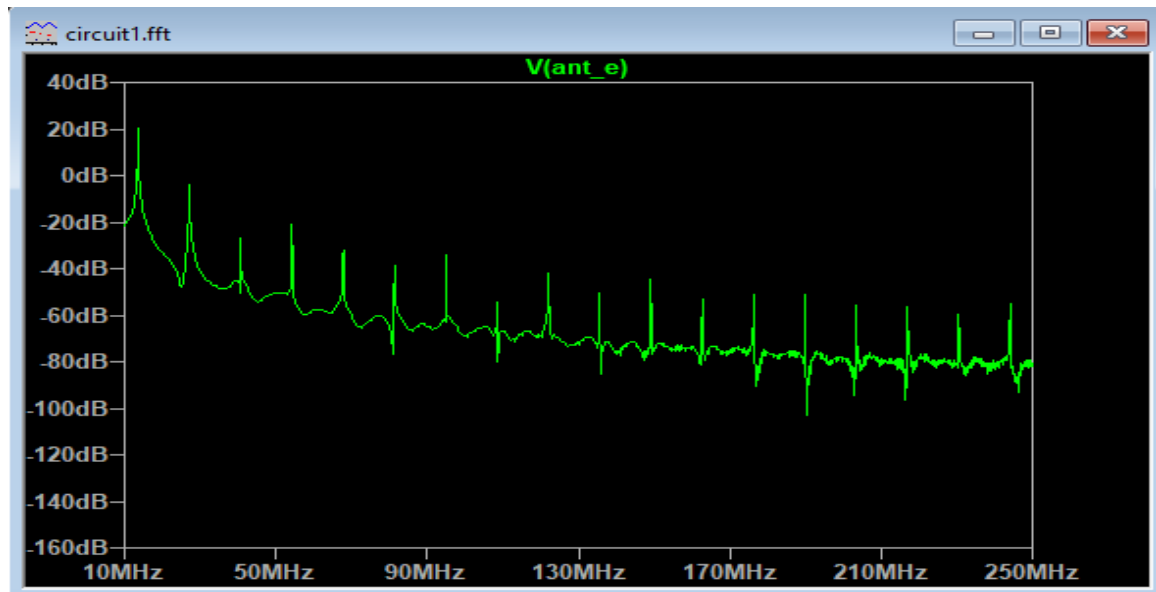


Figure 68 FFT of CC Plasma Varactor Equivalent Circuit 10 MHz to 250 MHz $V(\text{ANT})$

The calculated FFT for the current flowing in the resistive element for the plasma ohmic equivalent resistor R1 ($I(\text{R1})$) is shown in the figure (Figure 69) below.

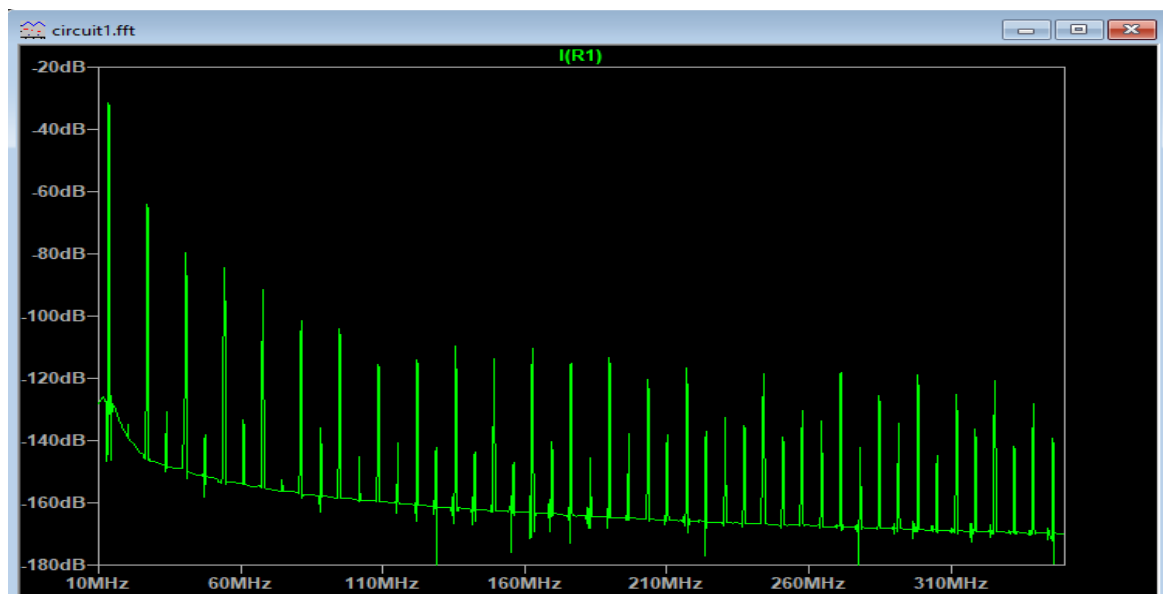


Figure 69 FFT of CC Plasma Varactor Equivalent Circuit 10 MHz to 350 MHz $I(\text{R1})$

This circuit demonstrates an equivalent harmonic content with the measured spectrum analyzer recorded output on the Oxford Instruments PlasmaLab 100 CCP. This is for a simple Argon plasma with no adjustment for pressure or gas flow. We will examine later the waveform that gives rise to the frequency components and how this changes dramatically for different gas flows and chamber recipes. By having a relatively simple model circuit that may be calculated in real-time that is tool-side, diagnostic information may be extracted from the processing recipes as they are executed on the tool.

In Kelly and McNally (Kelly & McNally, 2017), the 5th, 6th, 7th and 8th harmonics have increased gain. This can be seen from the plot below of the observed B-field.

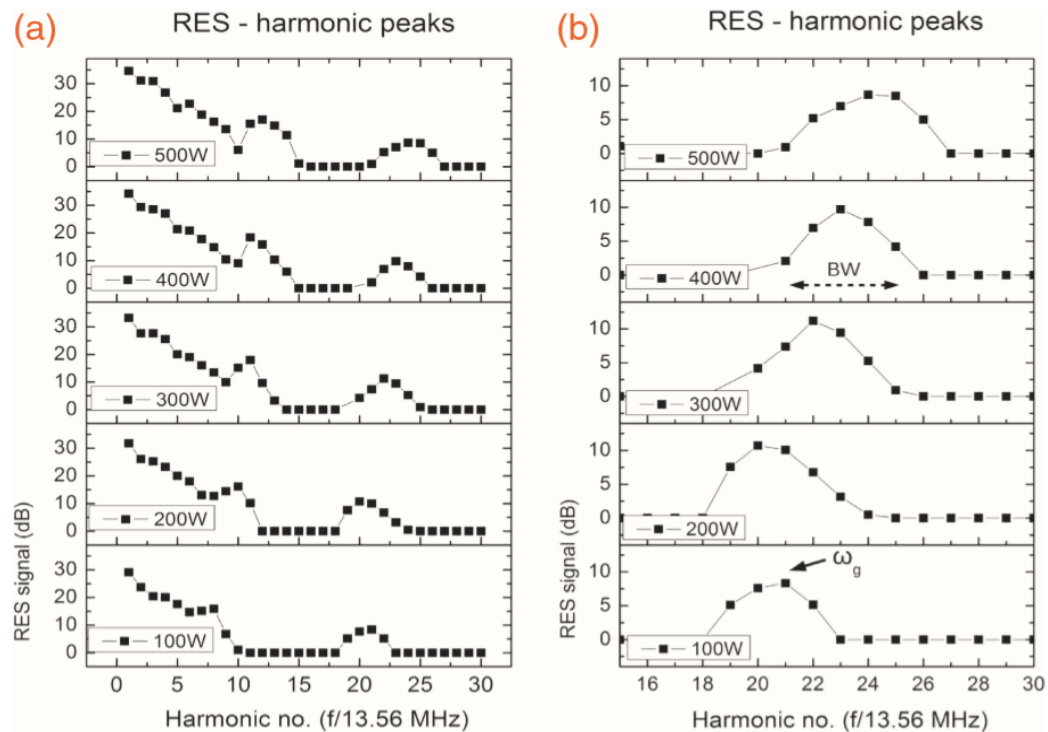


Figure 70 RES Measured B-field Response Curve (Kelly, McNally),

The increased intensity of these harmonics has been attributed to a geometric resonance associated with the physically smaller grounded sheath. The larger high-voltage sheath was correlated with higher frequency geometric resonances, which were roughly located around the 20th to 25th harmonics (Kelly & McNally, 2017). We plot the simulated FFT output of the current in R1 in a similar manner to the graphs in Figure 70 to compare the simulated results with that of the measured B-field curve.

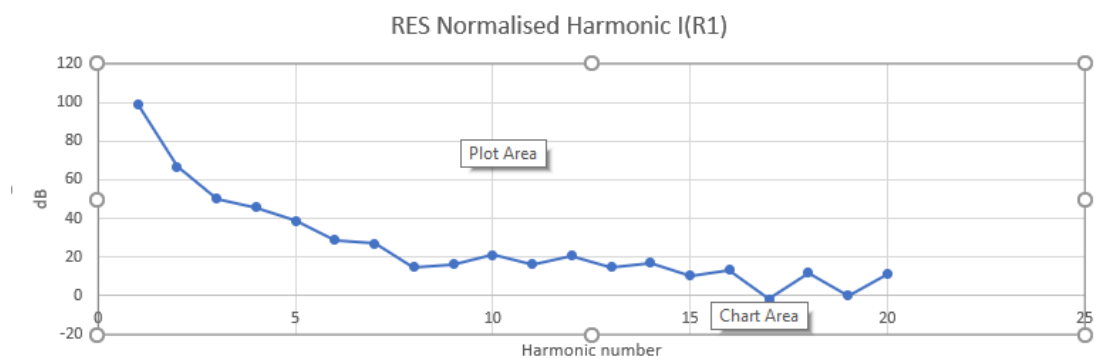


Figure 71 Simulated Harmonic Content Equivalent Circuit

We see the monotonic decay of the harmonics, until harmonics 5-8 of the fundamental of 13.56 MHz are reached. At this point there is a slight increase in the measured current. For a given pressure, the impedance of a discharge decreases with increasing frequency (Chapman, 1980). If the frequencies are being generated by the virtual diodes in the equivalent circuit and the mixer function is generating these frequencies, we can simulate these by resonant circuits at these harmonics. We add resonant circuits to model the increased current for these by loading the current flow through the bulk resistance to ground. This is only a simplification to increase the current flow based on fixed value resonances of the fundamental. This is scalable for the multi-harmonics that

are recorded. The circuit below in Figure 72 simply shows the tuning for the first 8 harmonics, but parallel units could be added to match the number of recorded harmonics. The following circuit shows these tuned circuits. By the addition of cascaded lumped circuits that are tuned for the particular harmonics by being resonant at the harmonic frequencies, the bulk current resonates and this is reflected in the measurement of the current through bulk ($I(R1)$). When the FFT is plotted, this resonance boosts the current flow through the bulk. The relative values of the tank circuits may be adjusted or additional resistors installed in series with the tanks to modify the specific harmonic.

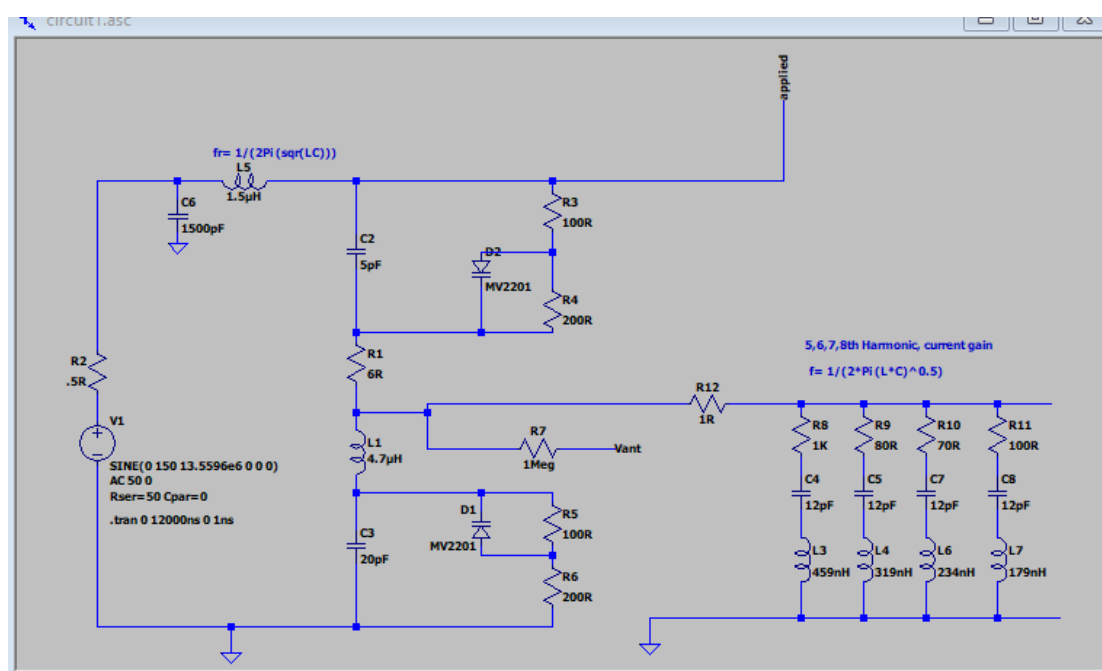


Figure 72 Tuned Harmonic Equivalent Circuit for CCP

The time response simulated output for this tunable circuit is shown in Figure 73 below. In (Kelly & McNally, 2017), the observation of two distinct resonance frequencies is due to the significant size difference between the driving electrode size and the grounded electrode in the PlasmaLab100 tool. The lower-voltage

grounded sheath contributes to the lower-frequency geometric resonances and the high-voltage driving electrode sheath to the high frequency geometric resonances. In the equivalent circuit it is possible to modify the Q of each individual harmonic. As given by (Kelly & McNally, 2017) *the series resonance bandwidth is given by the ratio of the resonant frequency to the quality or 'Q' factor* (as shown in the figure (Figure 70) above). There is a direct relation of the bandwidth (BW) to the electron-neutral collision frequency. The equivalent circuit permits the model to be able to set the Q factor by adjusting each individual harmonic gain and thus set the expected electron-neutral collision rate for a particular operating mode of the plasma under observation.

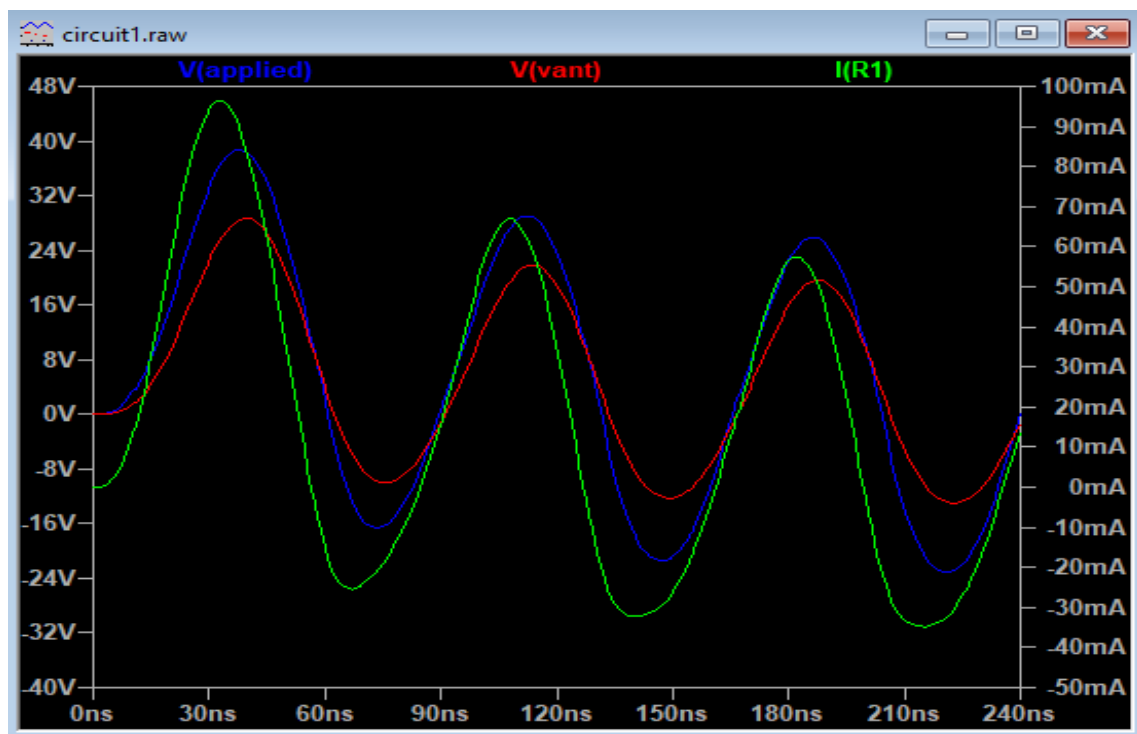


Figure 73 Time Response Simulation for Tunable Varactor CCP Equivalent Circuit for Plasma Bulk Current $I(R1)$

In this plot we see the non-sinusoidal waveform of the $I(R1)$ representing the bulk current. Also, in the plot is the applied voltage to the plasma. The circuit is started

from time $t = 0$ so the initial charging of the capacitors and the establishment of the bias is being shown. The FFT is taken of the current in R1 representing the bulk current (See figure 72).

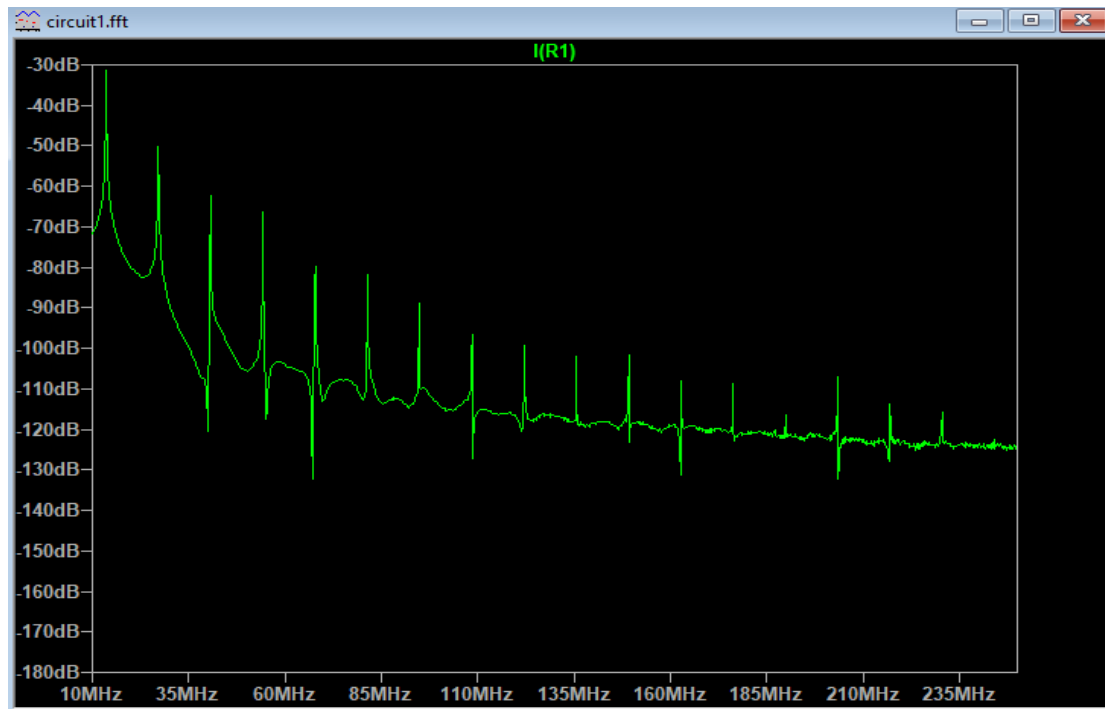


Figure 74 FFT of Tunable Equivalent Circuit of CCP for Bulk Plasma Current

The cascaded tank circuits may be tuned to modify this current and this is reflected in the power measurement of the various frequency high order harmonic components. We plot the normalized FFT harmonic values for this tunable equivalent circuit (see figure 72) and compare with those as observed in Figure 70 (Kelly & McNally, 2017).

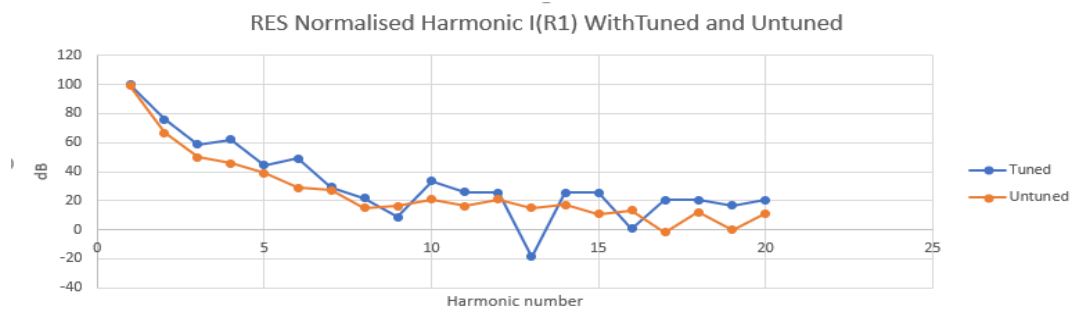


Figure 75 Simulated Harmonic Content Tuned Equivalent Circuit for Bulk Plasma Current (comparison to untuned circuit model)

We see a better match to the initial harmonic monotonic attenuation with the characteristic rise in signal for the boosted tuned harmonics. Further work is required to calculate the cascaded filters to model the change in bandwidth of the geometric harmonic for particular operating modes and recipe parameters of the operating plasma in the chamber. This model may be pre-calculated for observed operations of the plasma and its recipes. The tool-side equipment may then observe the correct operation of the plasma parameters to determine if a production process is operating correctly.

5.5. Hot and Cold Electron Jets Simulation

With the circuits shown, the diodes are used to introduce the current modulation for the non-sinusoidal harmonic generation. In the work of Wilczek *et al.* (Sebastian Wilczek, 2016), it appears that the modelling of the hot and cold electron jets in different directions is not included. For the hot electrons travelling in the 'reverse' direction, we introduce a simple voltage source with a source resistance, a pulse output for one quarter of the RF drive cycle time but with a phase delay. The following diagram (Figure 76) shows the circuit.

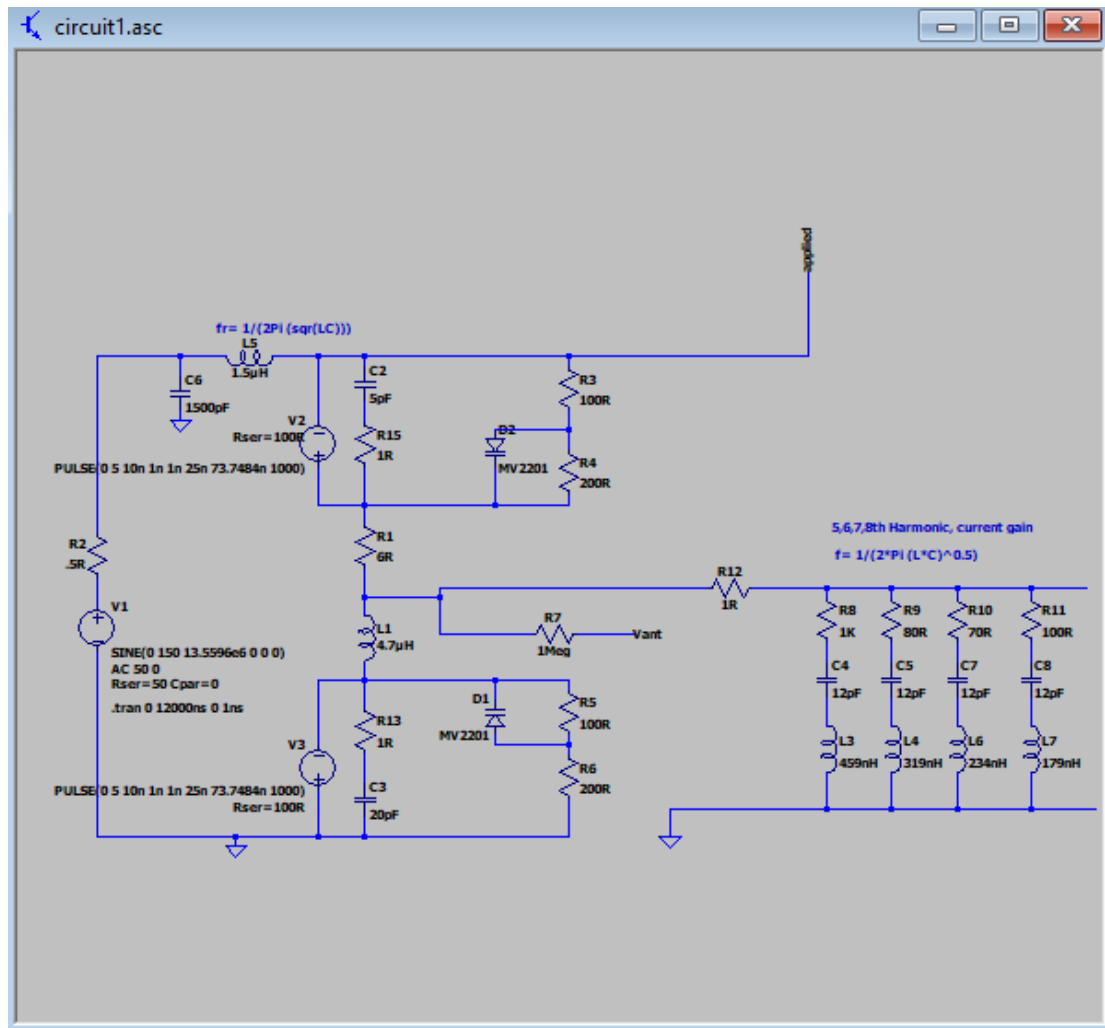


Figure 76 Equivalent Circuit of CCP with Provision for 'Reverse' Hot Electrons

By varying the source resistance of the reverse voltage source, it is possible to introduce the current spike that is seen when recording the B-field using the high-speed scope. This is shown in Figure 77 below, where the Agilent AG 80000 10 GHz scope was used to acquire signal from the custom plain coil antenna and the Beehive E-field antenna on two channels.

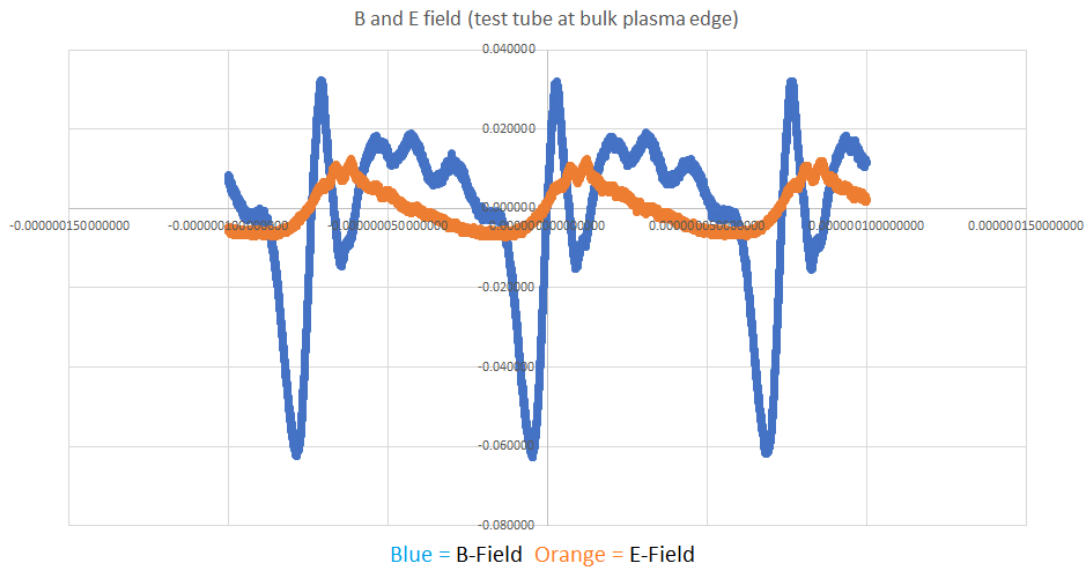


Figure 77 B-field and E-field High-Speed Acquisition using Agilent 40 Gs scope for TRES

The reverse current spike can be observed in the waveform for the B-field as the RF cycle is applied.

The following diagram (Figure 78) shows the time-resolved simulation of this circuit.

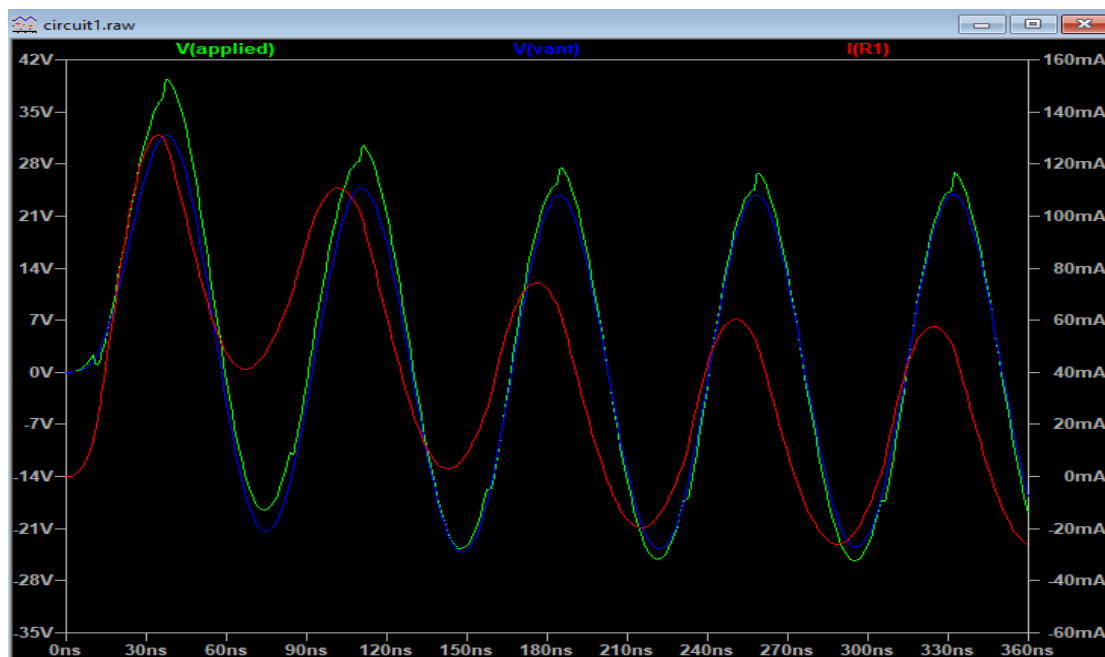


Figure 78 Equivalent Plasma Circuit Time-Resolved with 'Hot Reverse' Electrons

The FFT of the bulk resistance current $I(R1)$ was captured and this is plotted below in Figure 79.

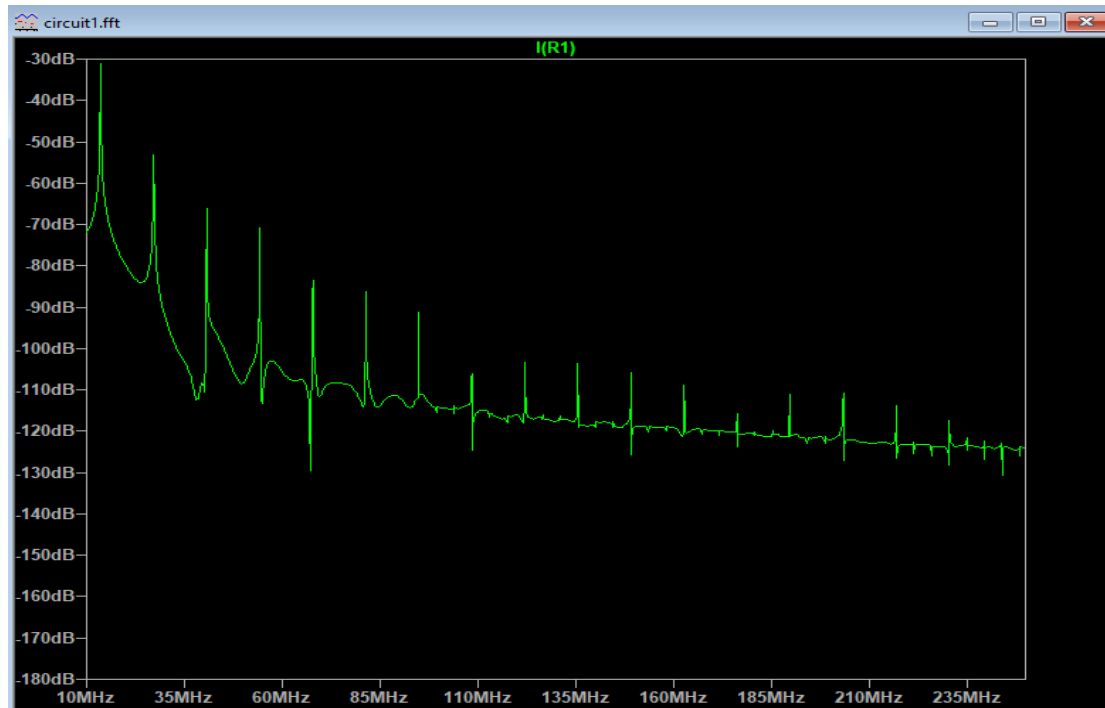


Figure 79 Simulated FFT Output of Plasma Equivalent Circuit with 'Hot Reverse' Electrons

The circuit shows significant boost to the lower-order harmonics over the circuit without the 'reverse hot electrons' voltage sources.

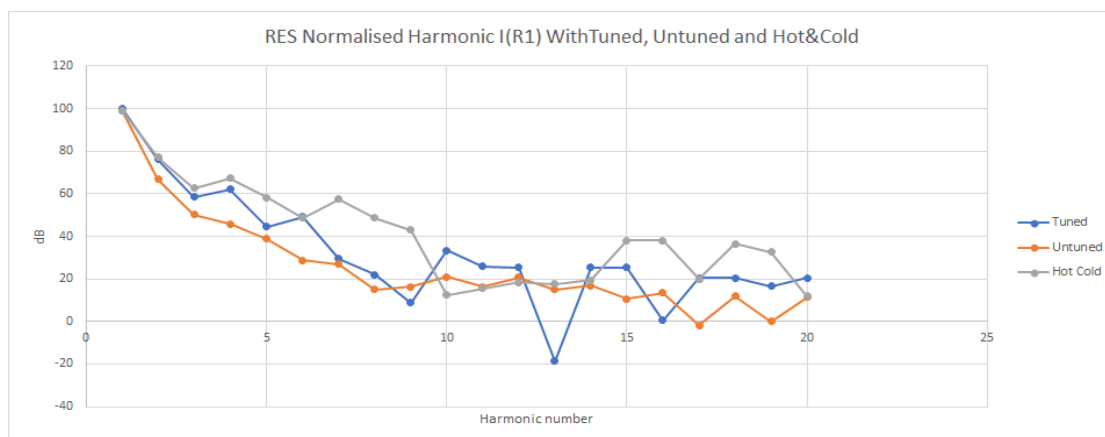


Figure 80 Comparison of Normalized Harmonics for Equivalent Circuit with Tuned Elements and Reverse Phase Currents for Hot & Cold Electron Beams

We then compared the simulated results with the measured harmonics for the Oxford Instruments PlasmaLab 100 CCP tool running a recipe of Ar plasma at 30 mTorr with O_2 at 20 sccm. We apply a scaling factor and offset to account for the measurement absolute values. This is a linear correction and can be accounted for with a combination of resistive dividers in the equivalent circuit. The following comparison is obtained. See Figure 81 below.

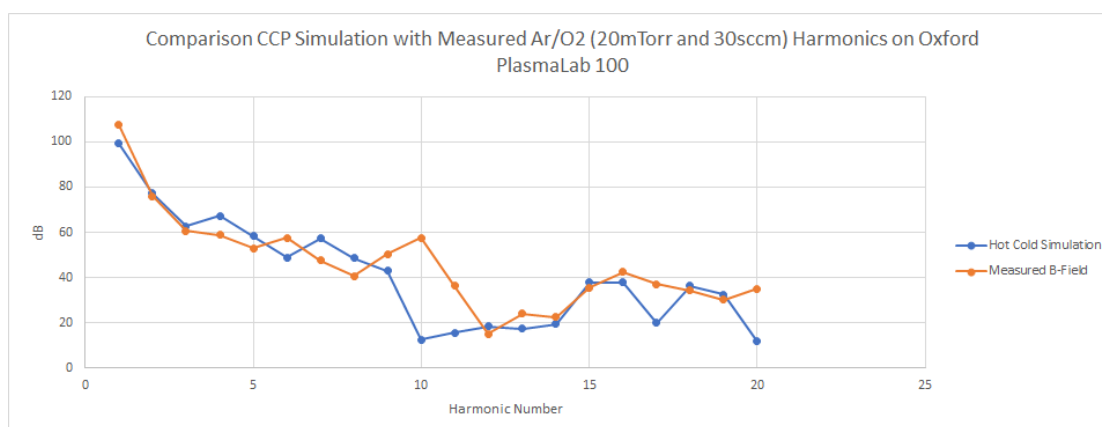


Figure 81 Comparison of Measured Harmonics and Simulated Harmonics with Varactor, Tuned Elements and Single Shot Phased Hot and Cold 'Reverse' Currents

Further work beyond the scope of this thesis will be required to simulate the range of voltages, time delays and source impedances to more properly match the operation of the test setup of the Oxford Instruments PlasmaLab 100, and indeed to generalize this to other plasma processing tools.

5.6. Improved model for Geometric Resonance and Hot & Cold Electron Injection

To further improve the dynamic nature of the operating capacitively coupled plasma, we present a novel circuit that more closely mimics the operation for the hot and cold electrons that are observed in the sheath expansion and contraction phases. The basic operation is to insert current sources in parallel with the non-linear diode and sheath capacitance and set the timings for these to match the spatio-temporal values as simulated by Wilczek *et al.* (Sebastian Wilczek, 2016). (See Figure 63 above). For the particular circuit, the values for the rise, sustain and fall of each ‘electron jet’ was estimated from the plot in Figure 63. Each section of density of electron jets is simulated by a current source with the phase, timing and intensity extrapolated from the plot with adjustment for the 13.56 MHz timebase of the Oxford PlasmaLab 100 tool. To further improve the geometric resonance phenomena as observed by (Kelly & McNally, 2017), an eight-pole passive bandpass filter is inserted to load the plasma and enhance the current for the geometric resonance. We also balance for the two-sheath configuration and insert a dc-bias load to control the effects of rectification that occur in the simulation. In the simulation, with the simulation components, the plasma bulk continues to charge through the diode charge-pump. In the operation of the plasma, the production of the ions and electrons stops once the current stops flowing and the sheath disappears. We model the ‘load-resistor’ as the work being performed by the plasma. This ‘load-resistor’ discharges the bulk-plasma and controls its plasma potential. The geometric resonance is modelled as a bandpass-filter centred around the geometric resonance frequency and additionally loads the plasma. The plumes of hot and cold electrons are extrapolated from the timing as shown by

Wilczek *et al.* as shown in Figure 63 taking into account the 13.56 MHz timebase of the Oxford Instruments PlasmaLab100 tool. We set the current to the 10 % of the peak bulk current and adjust for the density calculated. The rise and fall times are calculated from the density rise-times, and the hold-times are calculated from the period of the density pattern. We mirror the current for both electrodes, but allow for the adjustment due to electrode asymmetry with voltage adjustment resistors which change the applied voltage into the sheath. The matching unit is not modelled and the ‘losses and stray parasitic’ load as modelled in Schmidt *et al.* (See Figure 60) is not included, as this represents a phase-adjusted parallel load to the plasma bulk. For process matching, we required a harmonic generator with geometric resonance to build a model to match the operation of the plasma on the Oxford Instruments PlasmaLab 100 CCP unit. The following shows an overview of the circuit components.

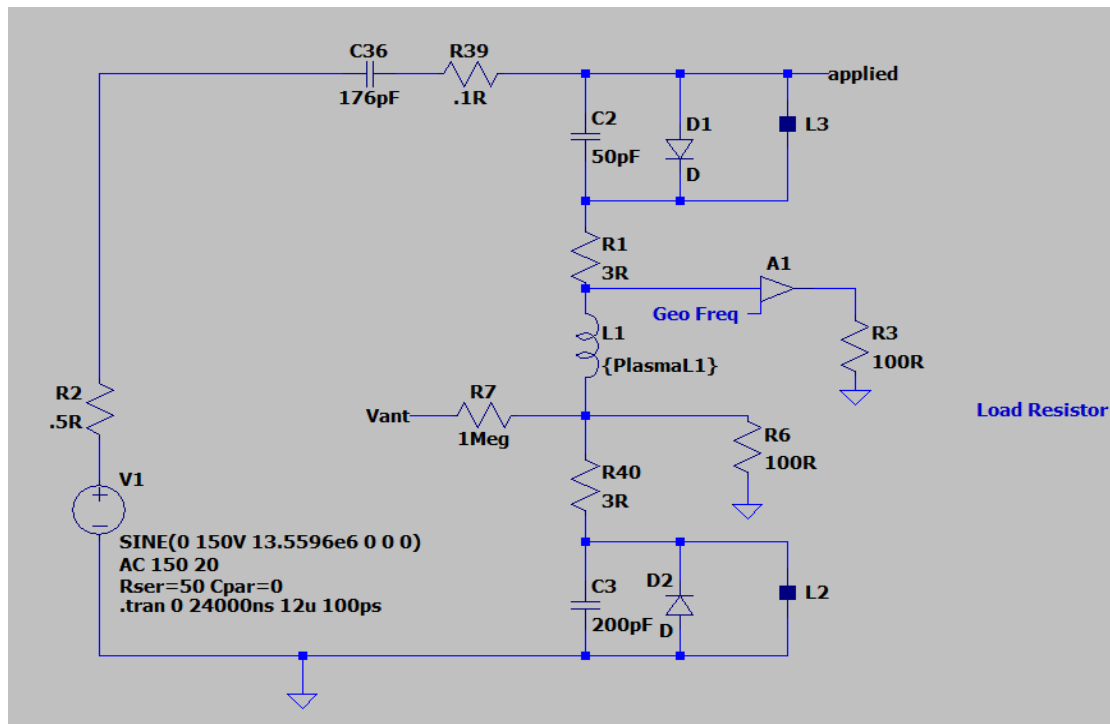


Figure 82 Geometric Resonance with Hot & Cold Electron Injection Simulation Overview

The variable units **L2** and **L3** as depicted in the circuit in Figure 82 represent the current generators for the timed injection of electrons that are in the sheath region as per the simulations of Wilczek *et al.* (Sebastian Wilczek, 2016). They can be seen in the full schematic in Figure 84. The unit marked **A1** is a calculable 8-pole RLC filter with an impedance of 100 Ω . We utilized a Chebyshev bandpass with a maximum of 3 dB ripple in the passband. The transfer function is shown below in Figure 83. This parallel combined load with the dc-load resistor (R6) provides the work-load for the simulation.

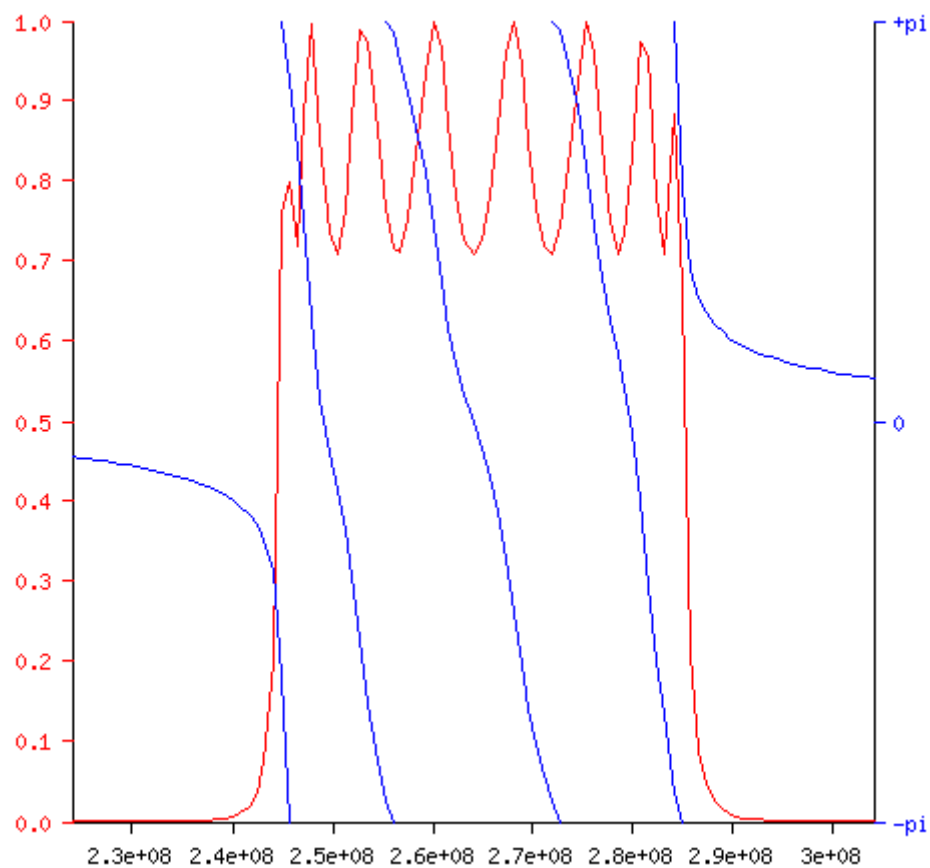


Figure 83 Normalized 8 pole Chebyshev Bandpass Filter for Geometric Resonance Simulation (Red is gain plot and Blue is phase plot)

The Chebyshev filter was chosen for the multiple phase reversals across the passband so as not to attenuate any particular harmonic in the time-domain analysis in the simulator. The complete schematic is shown below in Figure 84, and the circuit listing is in Appendix U.

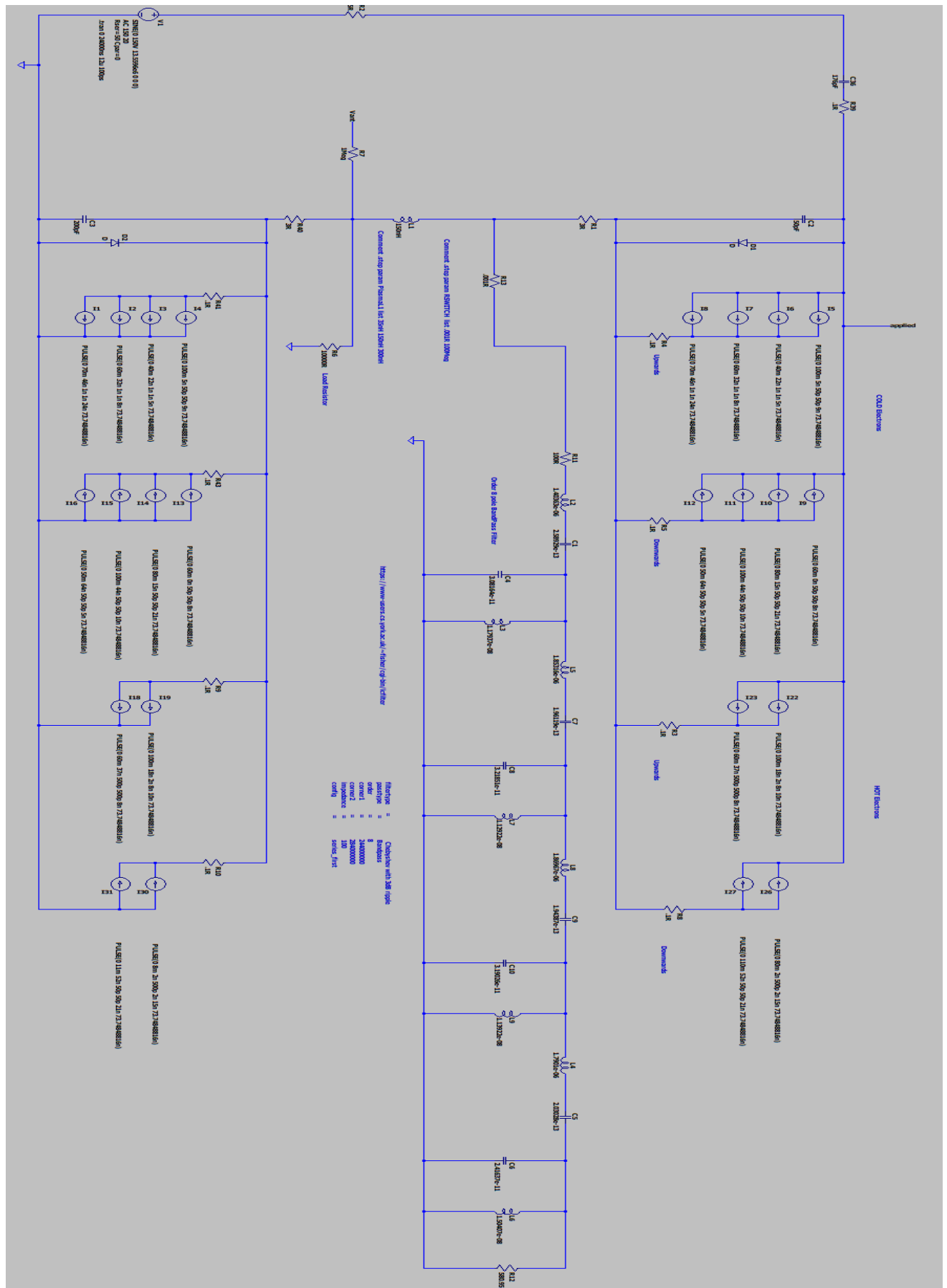


Figure 84 Geometric Resonance with Hot & Cold Electron Injection Circuit

When we run the simulation of this circuit, we can model the harmonic generation that occurs in the sheath of the plasma and also the geometric resonance location. We note that the impedance of the filter remains relatively constant for the range of frequencies produced by the non-linear sheath elements (the diode and the simulated hot and cold electron currents). We may adjust the filter coefficients to adapt the harmonic geometric resonance to match those of the operating plasma. Of note in the simulation is that we measure the harmonic current through the plasma bulk. We must be careful not to attenuate the harmonic generation in the sheath too much by having large amounts of inductance in series with the sheaths as this has a damping effect on the harmonic generation. (This can be used to model increased power as the harmonics fall off as the power is increased.) This is modelled with the '*step*' function in the simulation to execute simulations for the varying values of inductance. The following is the harmonic spectra with and without the geometric simulation model.

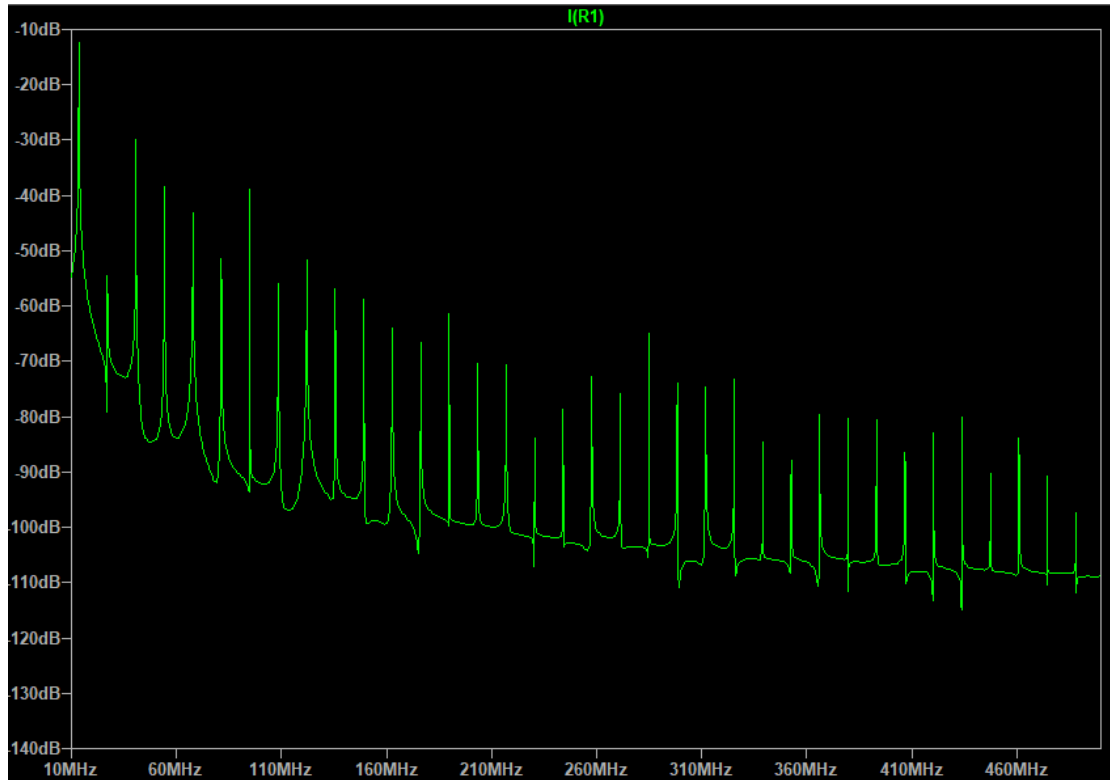


Figure 85 Spectrum Response from Hot and Cold Electron Injection Model with Characteristic Geometric Resonance

We take the frequency harmonic components generated by the simulation model and apply these to the filter system to test the applicability of the bandpass filter model for incorporation in the electrical circuit simulation. We extract the first 30 harmonic values, convert these to linear measurements and build a circuit to test the function of the simulated geometric resonance filter.

The circuit for this test simulation is as follows (see Figure 86 below):

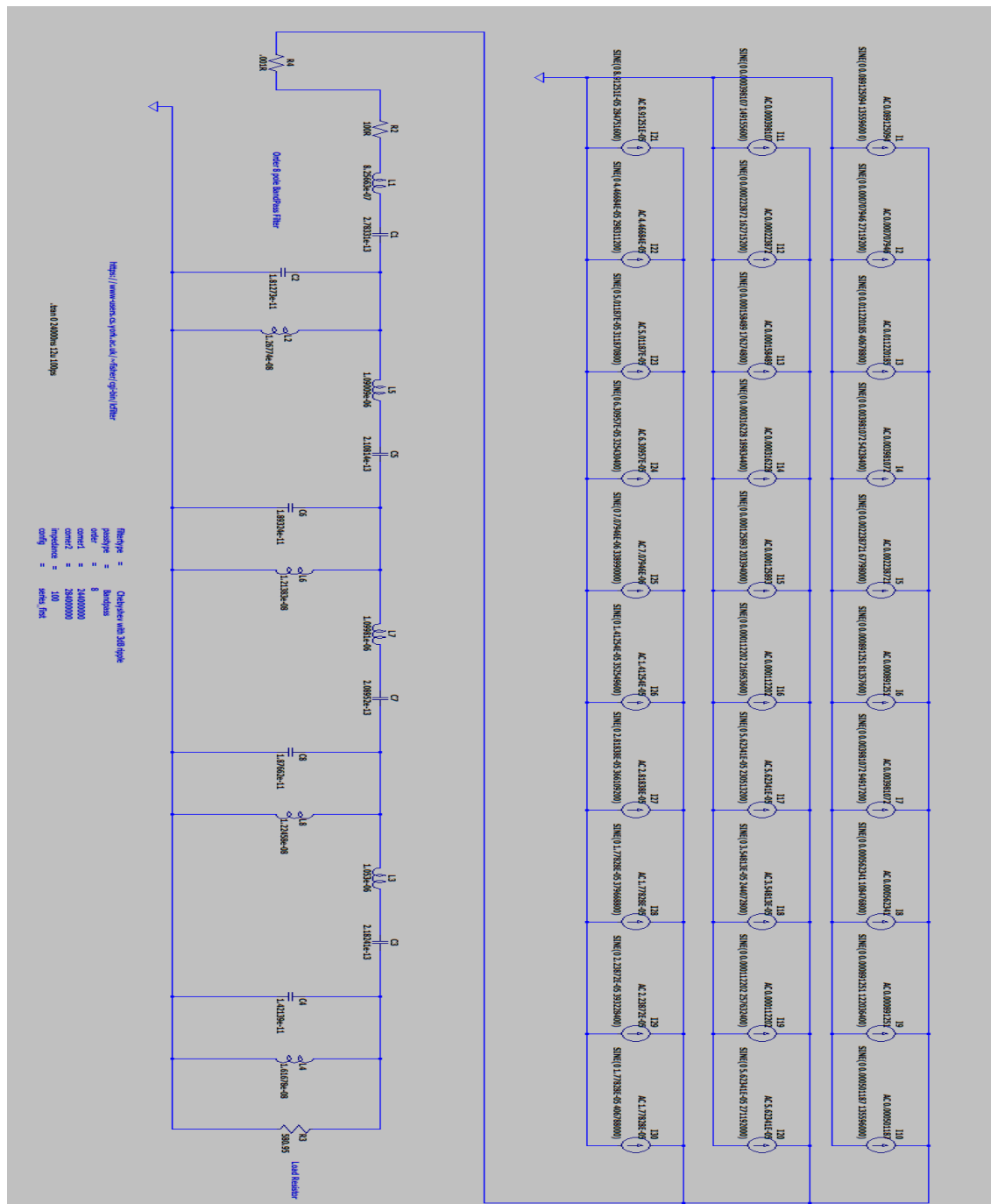


Figure 86 Eight Pole Geometric Resonance Simulation Test

This produces more than the required bump (+70dB) for the harmonics as observed in Kelly/McNally (Kelly & McNally, 2017) and may be suitably attenuated in the overall model. The harmonic output of the filter is as plotted below (see Figure 87).

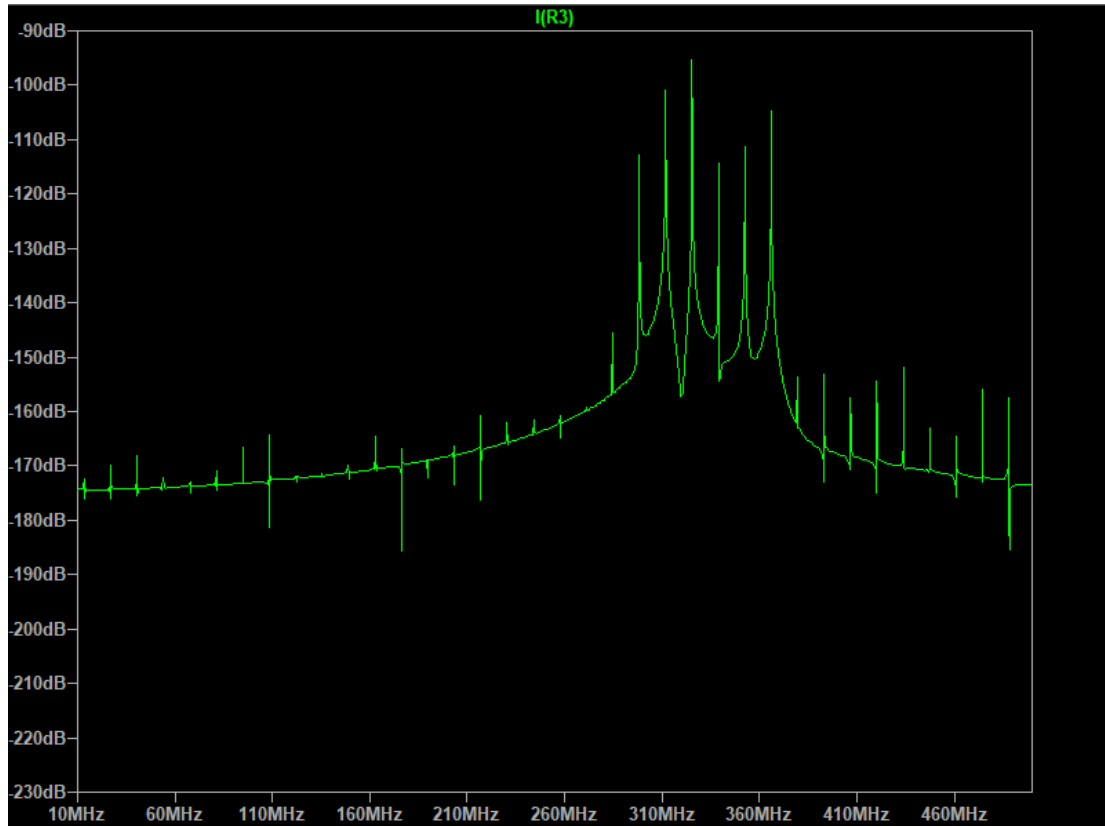


Figure 87 Geometric Resonance Harmonic Output with Simulated Plasma Harmonic Drive

We then plot this is in the familiar harmonic bar-graph to show the characteristic geometric resonance as observed by Kelly/McNally (Kelly & McNally, 2017). The geometric resonance as modelled by the eight-pole bandpass filter with its calculable corner frequencies. The model bandwidth can be matched to the bandwidth of the resonant frequencies in the higher harmonics for increasing power in the plasma. *“Damping of the series resonance is caused by the electron–neutral collisions given by the frequency ν_m [see Eq. (8)]3) and is constrained in the upward direction by the driving frequency ω (i.e., $\nu_m > \omega_{rf}$). Continuing our RLC circuit analogy, the series resonance bandwidth (BW) is given by the ratio of the resonant frequency to the quality or “Q” factor $\omega_{geom} = Q$, where the Q factor is*

found using the relation $X_C=R$ or $X_L=R$ for the capacitive reactance $X_C = 1/\omega C$ or inductance $X_L = \omega L$." Kelly/McNally (Kelly & McNally, 2017).

There is a direct relationship of the bandwidth of the geometric resonance harmonics to the electron-neutral collision. In this model, the eight-pole bandpass filter controls the spread of the series resonance at the geometric resonance. We adjust the poles of the filter to control the gain of the harmonics. In this case we plot for two different filters (8pole Number 1 and 8pole Number 2). We can see how the harmonic resonance is attenuated for Number 2 and there is gain for Number 1. For reference purposes, we have plotted the normalized 50 millitorr 20 sccm O_2 spectra as recorded in the experiments for Chapter 4. The filter poles may be adjusted to match the recorded geometric 'bump' as observed in Kelly & McNally.

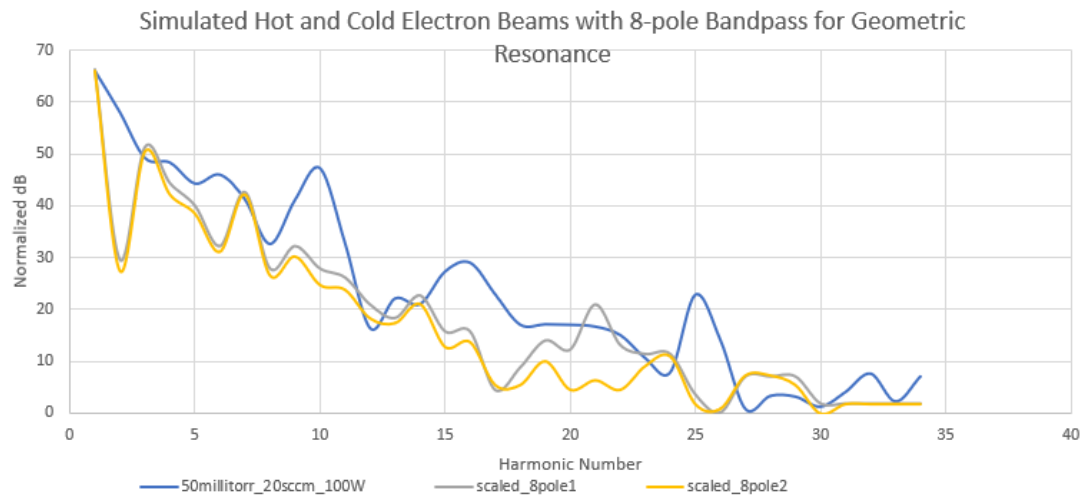


Figure 88 Simulated Hot and Cold Electron Beams with Characteristic Geometric Resonance Mapping

This extended model gives a tunable system to match the recorded spectra as observed with the recording of the B-field. By operating the plasma system with a

known steady-state and building a set of models to match the observed conditions, a calculable model may be built that may be used in the feedback control system of the Oxford PlasmaLab 100 tool. Other systems would require characterization and parameterization to build similar models leading to an improved empirical model of plasma systems. We can see that by tuning the bandpass filter we can match the 'geometric resonance' as determined by Kelly/McNally (Kelly & McNally, 2017) and also the individual harmonics may be set by varying the model parameters. While the match between the simulated data and the experimental data is far from perfect, it points the way to a new methodology. By operating the plasma system with a known steady state and building a set of models, the operation of the plasma in an industrial environment may be configured in a closed-loop mode. This will improve the accuracy and repeatability of the operating tools and thereby the output product.

5.7. Tool Side Monitoring Models

The monitoring of plasma tools being used in industrial plants must be fast, non-invasive and deterministic. Most industrial plasma systems are open-looped and the monitoring is of the static parameters, for example power being applied, gas-pressure set-point and gas-flow rates. By monitoring the high-order harmonics and matching those to pre-calculated equivalent circuit parameter values, the overall status of the process may be determined for event detection and non-conformance. These equivalent circuits may also be used for tool matching. In the next chapters, we will examine further transform methods to determine the correct operating mode of plasmas and event or anomaly detection.

Chapter 6 K-Nearest Neighbour & Radial Basis Function Modelling for Spectral Analysis

6.1. Introduction

K-Nearest Neighbour (KNN) is one of the more simple classifiers along with regression classifiers (Guo, Wang, Bell, & Bi, 2004). In Chapter 4 we have seen how the Wavelet Transform can be used to identify the instantaneous frequency components of the plasma system and we would like to use a mechanism to classify these instantaneous events. The primary focus here is on a practical implementation of the KNN that would be applicable to FPGA devices for classifying the FRES output or the output of the Hilbert or the Wavelet transforms in real time from the TRES acquisition. KNN can be used for both classification and regression predictive problems although the former is more prevalent (Bolandraftar, 2013 Vol3 Issue 5). It has a relatively inexpensive compute time and its output can be determined at a fairly quick rate with a compute time of $O(n^2)$ for a space complexity of $O(n)$, if the space is maintained at a modest level. The hardware solution uses a parallelizable method and the results can be obtained on the fixed hyperspace array size in a deterministic manner. The basic function is to calculate the radial distance from all the points and find the points with the nearest locale. For example, the KNN hardware function of the Intel Quark® microprocessor running at 4 MHz can find the nearest point to a vector within a 128 X 8bit unsigned array at a rate of 25,000 per second. An Intel i7 processor doing it in the software presented running at 3.6GHz can only search the same hyperspace at a rate of 12,000 per second (see Appendix I). This is singly threaded, non-optimized non-parallelized code. This is of importance in the design

of a detector for arcing events within an industrial plasma chamber where events occur in only a number of RF cycles of the machine. With 13.56 MHz being a common drive frequency and events of the order of a few microseconds, the KNN would have to be evaluating input vectors approximately every microsecond.

6.2. K-Nearest Neighbour (KNN) Operation

The KNN algorithm is non-parametric (though the classifying vectors may make use of parametric values) in that it does not make any assumptions on the underlying data distribution. The operation uses a database in which the data points are separated into several separate classes to predict the classification of a new sample point. The model structure is derived from the data itself (Laguna & de Andrade Lopes, 2010). A subset of recorded data is taken (approximately 20% for practical purposes) and the classifiers are labelled and used as training data. This training data is then used to build a model describing the complete set of classifications. These classification labels are then used to populate the hyperspace. The remaining recorded data (perhaps 80%) is then run through the classifier and its classification is compared to its predicted classification and the accuracy of the model is calculated. This can be an iterative problem until the optimum model is derived.

For industrial use this allows it to be used when there is little or no prior knowledge of the distribution of the data. All the training data is used to build the classifiers and then the new data is compared to the classifiers to determine a classification. To decide whether observations taken at a different time are of a similar class, clearly the method of comparison is germane. This is the

classification problem. Even when we have decided on a method to compare two observations and can quantify their similarity, there is still the problem of deciding which observations from the database are similar enough to our new observation such as to take their classification into account when classifying the new observation.

6.3. Radial Basis Function (RBF) Network / Operation

The Radial Basis Function (RBF) network is one type of neural network and can be used as a non-linear classifier. As per the KNN algorithm, it is not too computationally intensive. It is similar in computational load to the KNN algorithm, of the order of $O(n^2)$. Each *neuron* in the Prototype Classification (as seen in Figure 89 below) takes a weighted sum of the input vector. This vector could be the input from the FRES harmonic detector, Wavelet Transform, the Laplace Transform or the other detectors described (and shown in Figure 89). Each input value is modified by a normalizer or preprocessor and fed to each Prototype Classifier. The output of the combination of the weighted prototype classifiers produces a category-score for each category. This combination permits a non-linear classifier (Goshvarpour, 2012). This is a single-instruction multiple-data problem (SIMD) and lends itself very much to a hardware implementation (Ahmed Shamsul Arefin, May 15 2014) . It may be implemented on FPGA for the classification of high-speed transient events for real time performance. The RBF network performs classifications by measuring the input observation similarities to examples from a training set. Each node of the network stores a particular labelled classification. For the classification of a new observation, the input vector is weighted or normalized and it is applied to all the linear Prototype Classifiers.

The Prototype Classifiers output values are then combined to produce a Category Score for each of the possible classifications. This is shown below.

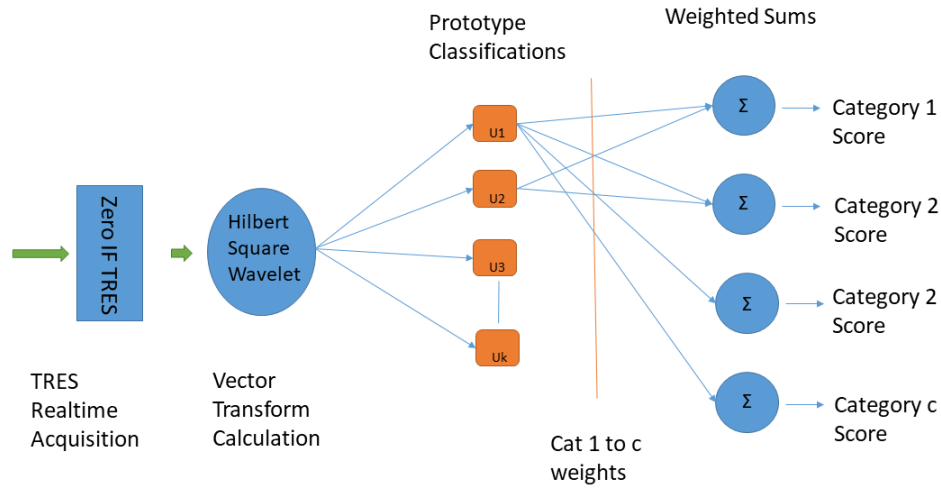


Figure 89 RBF TRES Event Neural Network Data Flow

The front-end acquires data and in this case, we are considering down-sampled zero-IF acquisitions. These may be transformed in a linear manner, e.g. by using a LPF, square law modulation detector, Hilbert or Wavelet transform to produce an *input vector*. The vector is applied to each of the prototype classifications. Each of these produces an output between 0 and 1 where the 1 is an exact match and 0 is no match with an exponential response. The response curve is nominally a Bell/Gaussian curve where the calculation is (Mehdi, 2016):

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean and σ the standard deviation.

Note that σ controls the spread of the normal curve and determines the categorization of the input vector for each neuron. Should the input vector match the neuron category exactly, the output will be at maximum.

It is possible to use other types of prototype classification functions such as the sinc function where:

$$\text{sinc}(x) = \begin{cases} 1 & \text{for } x=0 \\ \frac{\sin x}{x} & \text{otherwise} \end{cases}$$

This is referred to as the sampling function and is very similar to the windowing function that we see in the SDR spectrum analysis and as before, the multiplication in the time domain produces convolution in the frequency domain.

The outputs of the category classifiers are then weighted and summed individually which produces a category weighted classification. This output vector can then be applied to the KNN or RBF classifier to determine the observation from the continuous acquisition or used directly. The prototypes must be trained (selected) to produce the required Category Selectors. In the KNN method, we must define the hyperspace to contain the input vector and this hyperspace must be divided up into the various classifications. In the particular model built, it is 128 vectors of 128 unsigned 8-bit integer (uint8_t). In the RBF case, the number of neurons (prototype comparators) determines the hyperspace of the input range and the weighted sums contribute to the number of categories that can be determined.

6.4. Software Implementation of KNN Engine.

Early software development for investigation of the data acquired from a Nikon 712 Electron Cyclotron Resonance (ECR) plasma tool was developed on the Intel Quark® processor which included a General Vision derived Pattern Matching Engine (PME). This PME is a hardware implementation of a basic KNN device. This

engine is able to process 25,000 KNN searches per second in hardware with the processor running at its maximum clock rate of 4 MHz. It is designed with 128 vectors of 128 x uint8_t (unsigned 8-bit integer) values that describe the hyperspace. The software modelling developed here was adapted to process the RF spectrum analyzer data acquired using the SignalHound SA44B on the Oxford Instruments PlasmaLab 100 using the bespoke software acquisition tool, the differential cabled balun system and the chamber flush mounted B-field antenna within the prototype test-tube. The software was developed using Matlab® to process the data and the test-code for the real-time high speed KNN classifier is included in the appendix (See Appendix I). A similar approach is adopted for the Hilbert and Wavelet transform characterization. As we wanted a general-purpose system that could be implemented tool-side to analyze the data acquired from the FRES or TRES acquisition subsystem, a KNN system engine was coded in 'C'. This was used to process and test the design of a classifier for the event detection of the FRES/TRES system. The modelling software code for the KNN model is included in Appendix R. The software version of the KNN engine running on an Intel 64bit i7 processor at 3.6 GHz can evaluate 12,000 first-nearest-neighbours per second. The results of the performance monitor are included with the code in Appendix I. For an FPGA system, either a smaller number of multipliers and hyperspace dimension KNN or a feed-through RBF system is more suitable. Multipliers are expensive in gate count for FPGAs. The design would be to minimize the columns and calculate iteratively over the rows. Also, by setting the hyperspace to normalized uint8, the multiplications are scaled integers and are bounded. This requires reduced resources for routing and gate count permitting the use of smaller devices thus reducing cost. The software implementation permits the

testing and evaluation of the method for the captured data in a batch-mode processing and may be transferred to FPGA implementation either using bespoke development or tools such as Simulink®. We implement the algorithms and test the efficacy of evaluation process in the following sections.

6.4.1. Oxford Instruments PlasmaLab 100 KNN Modelling Engine.

To build a KNN model for the Oxford Instruments PlasmaLab 100 tool, a number of experiments were performed. This was a two-stage process, the first being to step the pressure using fixed flows of gas and the second was to keep the pressure constant while stepping the flow rate of gas. This permits the building of two distinct basis models for the linear classification of the operating parameters of the plasma. The FRES harmonics were recorded using the bespoke BBS recording software operating on the SignalHound SA44B with the patent-pending balun transmission system. The custom B-field plain-coil probe was inserted into the fashioned Pyrex® test-tube to operate parallel to the wall of the plasma chamber (see Figure 62 and Figure 64 above). In Chapter 5, the pictures including the chamber, the patented balun cabling system, the antenna and the test tube show the orientation of the apparatus to acquire the RF harmonics as the plasma is operating.

In the figure (Figure 90) below, the PC connected to the SignalHound SA44B running the developed application code acquires the individual harmonics and displays them as continuous graphs in time. On the PC screen, three harmonics are being recorded in time, with step changes being made to the plasma process.

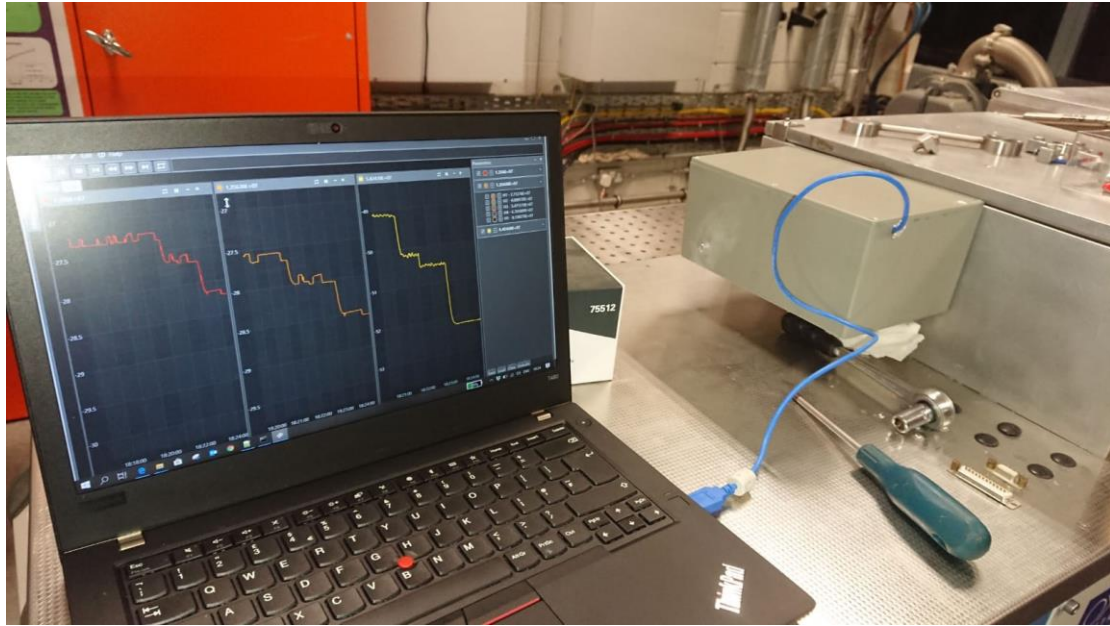


Figure 90 Screenshot Capture of Collection of FRES Harmonics with Plasma Recipe Variation

By inserting the B-field probe inside the test tube flush to the inside chamber wall (see Figure 64), a signal with a much richer harmonic content was obtained. The near-field FRES high frequency harmonics are greatly attenuated when recorded outside of the KF40 flange. The recorded data is used to build a multivariate dataset for the learning phase of the KNN models. There was no measurement of harmonics below the drive frequency. In addition to the harmonics generated by the non-linearity of the sheath operation, 'slow surface waves' may exist (Steven, Sowa, & Cecchi, 1996-01-01), these are of shorter wavelength than the drive. Owing to the complex interaction of the sheath non-linearity with the specific boundary conditions, this 'slow wave' may be observed. This 'slow wave' still has a frequency higher than the drive frequency. For example, in Steven *et al* (Steven, Sowa, & Cecchi, 1996-01-01) with their experiment at a drive frequency of 135MHz, the wavelength reduction was in the order of 5. The wavelength reduction is thought to occur as the E-field is confined to the thin sheath region,

and the magnetic field penetrates further towards a value equal to the RF skin depth, which can be much larger than the sheath thickness. This magnetic-field penetration increases the inductance per unit length of the equivalent transmission line, thereby slowing the wave. This is modelled as increased inductance that tends to oppose the applied field (Paul A Miller, 2006). There is a large spatial variation in the generation of the frequencies and the models did not take into account the probe position with respect of the electrodes. Further work is required to reposition the probe and build models taking this positional variation into account. The measurements were taken radially to the bulk plasma, flush to the surface of the grounded chamber (as seen in Chapter 5). See Figure 91 below.

The FRES measurements were captured up to the 103rd harmonic (1,396,646,794.0 Hz) of the drive frequency (13,559,677 Hz). Earlier studies only examined the RF spectral range up to 54 MHz (the fourth harmonic) (M J Colgan, 1994) and in the work of (Paul A Miller, 2006) a spatial variation analysis produced only the first 10 harmonics from the FFT analysis for a 13.56 MHz driven capacitively coupled plasma system. As we will show later, the examination via principal component analysis (PCA) confirms that the varying contribution of the higher order harmonics describes a complex plasma interaction to parameter changes.

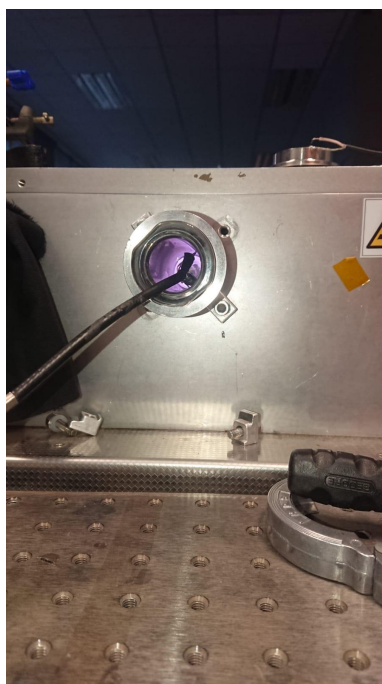


Figure 91 Antenna Location for FRES acquisition for High Harmonic Content

Two experimental regimes were utilized. These were modifying the pressure and keeping the gas-flow constant and the second was keeping the pressure constant and modifying the gas-flows. The experiments may be summarized as follows:

Experiment	O_2 Flow rate		
	20 sccm	25 sccm	30 sccm
Pressure			
20 millitorr			
30 millitorr			
40 millitorr			
50 millitorr			

Table 5 Experimental Cross Table

The FRES harmonics were recorded for each experimental regime. The data was tagged with the recipe parameters. An example of each harmonic recorded is shown below showing the narrow band frequency content with an acquisition

bandwidth of 250 kHz. The individual harmonics are narrowband and move with the base drive frequency. The drive generator for the Oxford Instruments Plasmlab100 is an Advanced Energy Hilight 136 13.56 MHz 600 W power supply shown in the picture below:



Figure 92 Oxford Instruments PlasmaLab 100 Tool RF Power Supply Unit

The power supply / signal generator is not very accurate and drifts significantly in frequency with load, time and temperature. The drift is of the order of 200 kHz when the power supply is first turned on until it settles to its steady-state frequency. The power supply frequency in the adjacent chamber on the machine is within approximately 500 Hz of the first chamber. This may be seen in the figure (Figure 93) below.

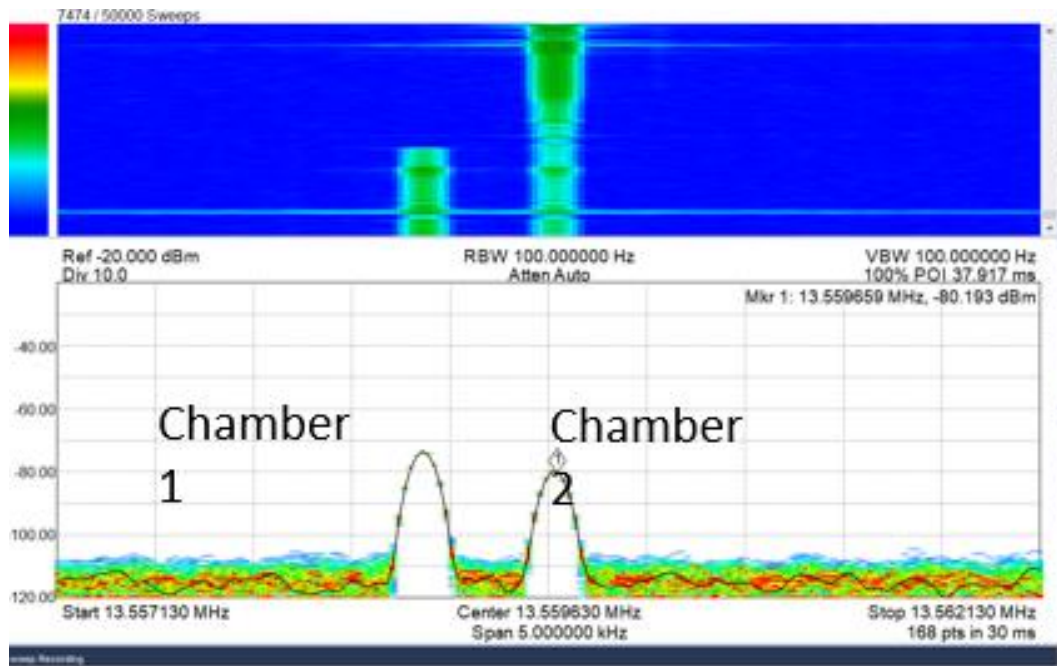


Figure 93 Adjacent RF PSU Frequencies on Two Chambers of Oxford Instruments PlasmaLab100 CCP Tool (screenshot taken from Spike™ software).

The primary drive frequency generates the harmonics which are very narrowband. To find the high-order harmonics, a bandwidth (BW) of 250 kHz around the estimated harmonic is recorded. Peak-value finding is then performed on these vectors of frequencies to extract the value of the particular harmonic. This peak value is then used for each recorded harmonic for further processing.

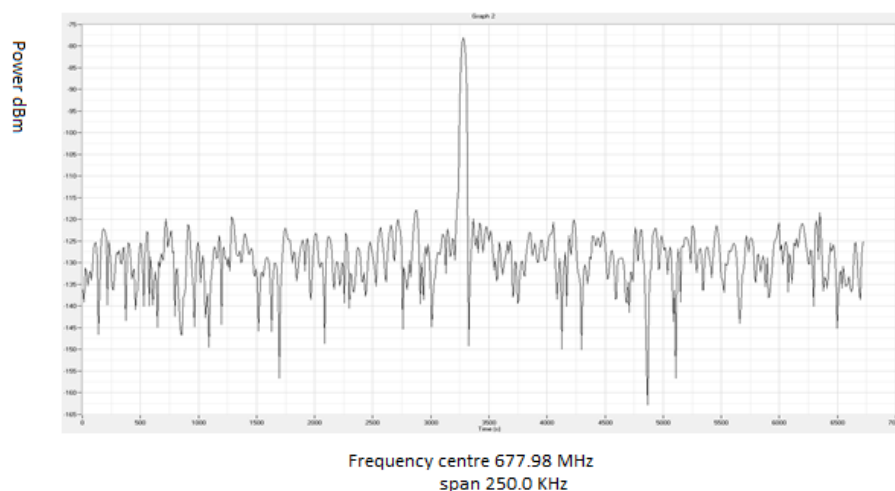


Figure 94 Recording of Example FRES Harmonic (50th of 13.5596 MHz) at 677.98 MHz

This is a narrowband acquisition with high resolution and accuracy. Based on the primary drive frequency, rounding errors in the high-order harmonic frequency calculation and errors in the calibration of the spectrum analyzer must be taken into account. Signals received by the B-field antenna are of the order of -80 dBm (22.36 microvolt) to 155 dBm (1.257 microvolt). These are recorded and tagged with their operating parameters. A subset of the recorded data is used to build the KNN model and the remainder is used to test the model and the results are then tabulated.

6.4.2. Experimental Data Set

The experiments consisted of recording the FRES harmonics on the Oxford Instruments PlasmaLab100 while adjusting the pressure and the gas-flow for O_2 in an Ar plasma. The pressure was adjusted for 20, 30, 40 and 50 millitorr and the gas flow was modulated from 20, 25 and 30 standard cubic centimetres per minute (sccm). Sufficient time was allowed for the plasma to settle and a suitable number of recordings of each of the 102 (primary was not recorded) harmonics were taken. Within the limits of the SignalHound SA44B, each harmonic requires approximately 300 ms to record and to complete a sweep of ~ 100 harmonics requires 30 seconds. Three to five recordings of each harmonic at a particular experiment steady state were recorded. This necessitated making the parameter change to the plasma and then recording for approximately 2 minutes at each step.

Some preliminary statistics of the results were first generated to evaluate the linear KNN model usage and the visualization of the FRES data. The data was

analyzed over the evaluation of the pressure change with the three gas-flows kept constant. This permits the visualization of the data in the pressure regime. The following graphs show the three experiments side-by-side.

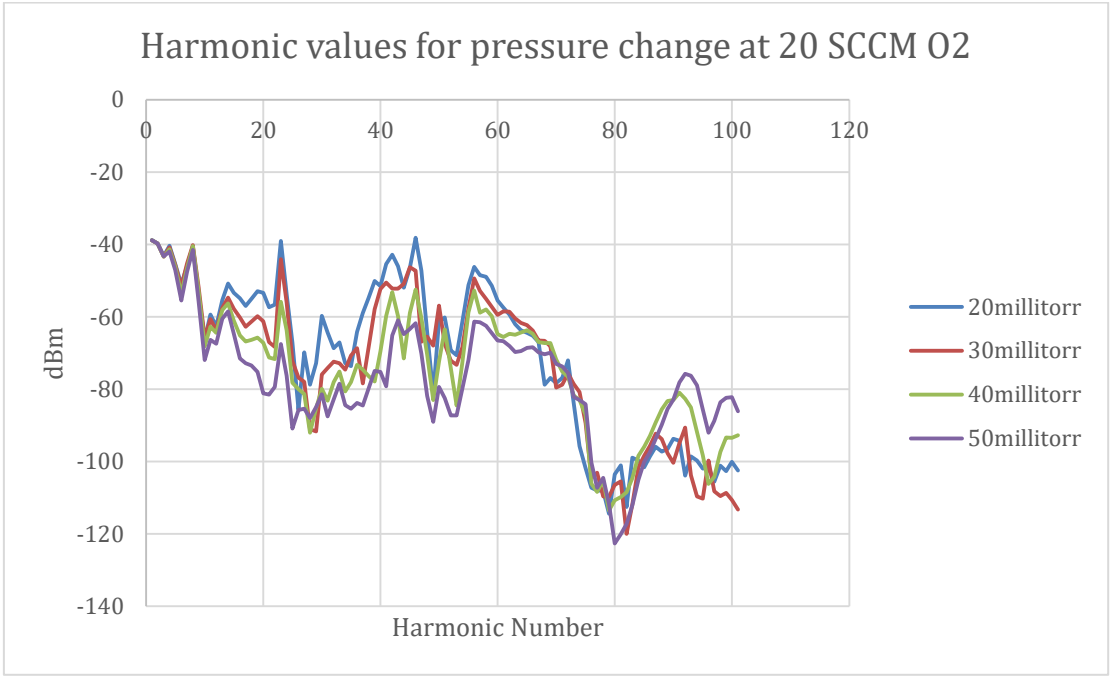


Figure 95 Plot of RES Harmonics as a Function of Varying Chamber Pressure

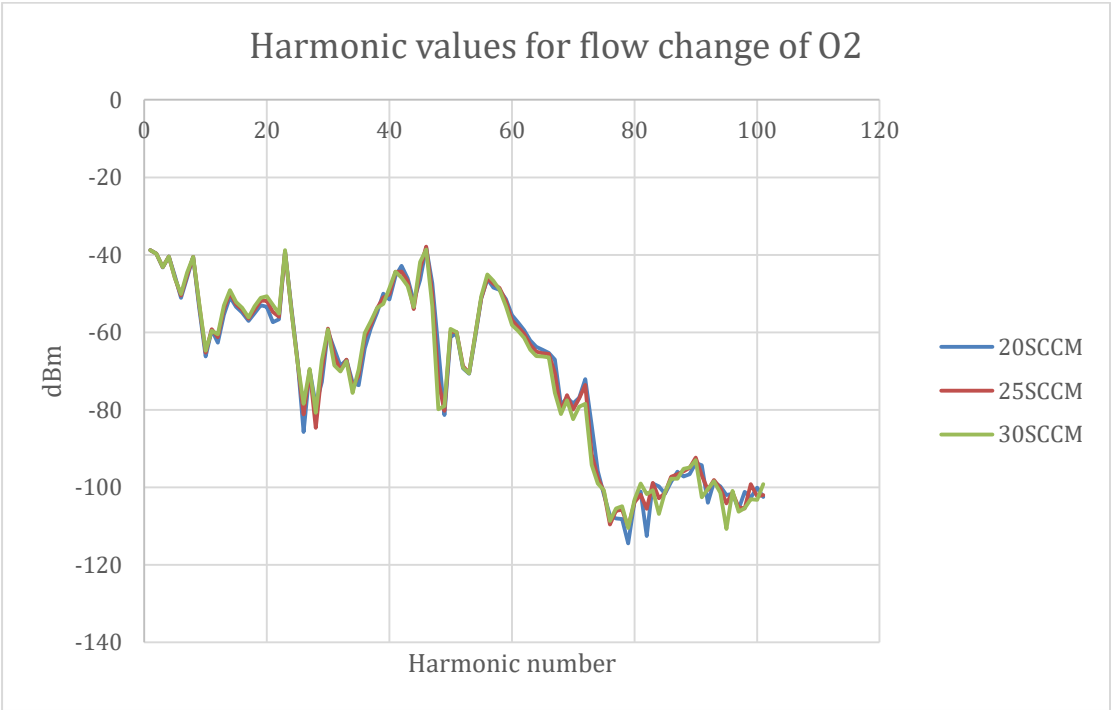


Figure 96 Plot of RES Harmonics as a Function of Gas Flow Change

As can be seen, there is little or no change to the harmonic content based on the flow rate of the O_2 whilst there is significant change for the pressure changes. We will see this later in the KNN results.

6.4.2.1. Harmonic Surface, Box and Principal Component Plots

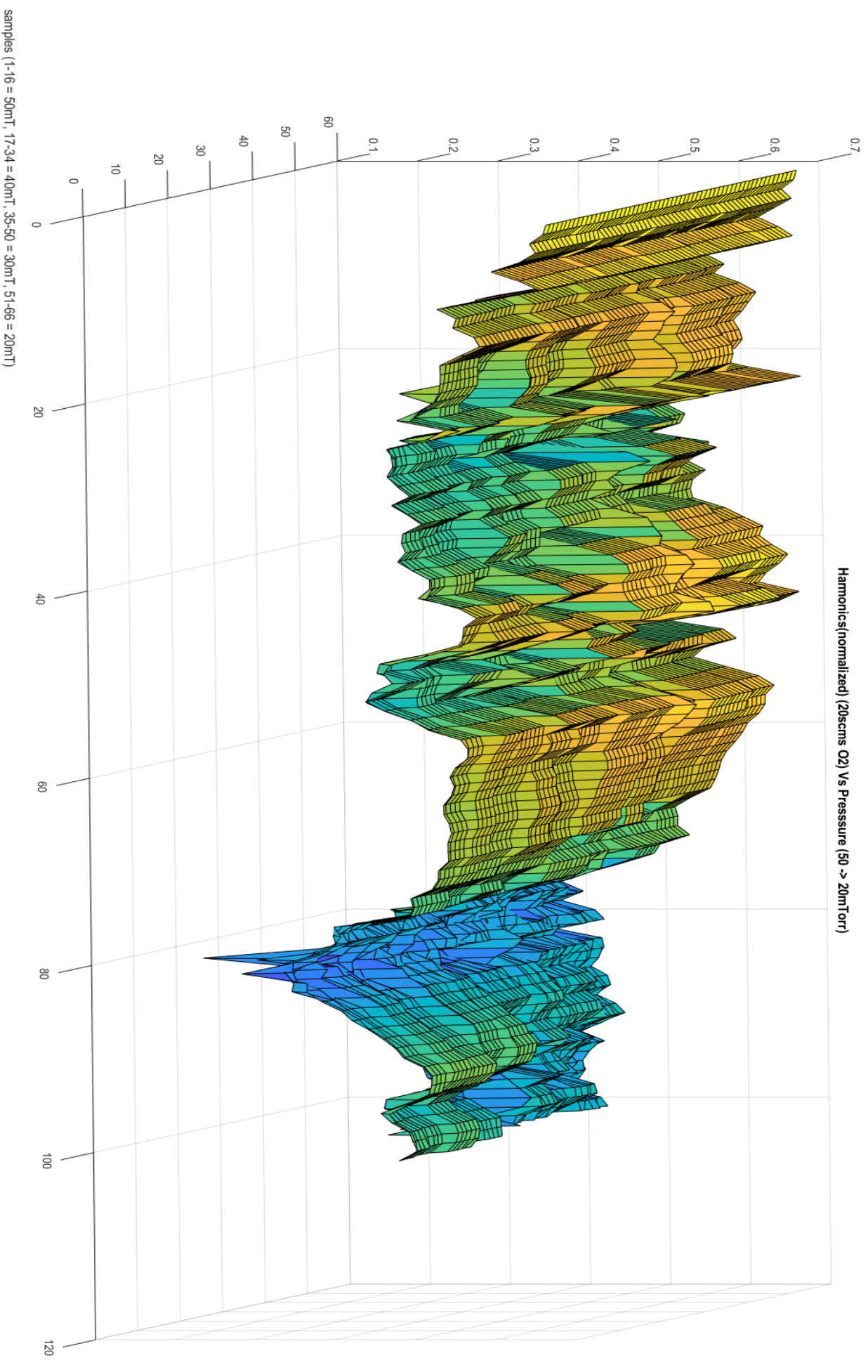
By plotting the harmonics (normalized) as a function of pressure and comparing for the different gas-flows, the same common feature set can be observed. The following figures (Figure 97) show the plane of harmonics as the pressure is varied. Up to 5 harmonic measurements were made at each 'steady state' mode of the plasma. The steady-state mode was a change to a particular pressure with a constant flow rate or a change to flow rate entering the plasma while maintaining a constant pressure. We graph the harmonics as a 'plane' with the change in pressure perpendicular to the page on the Z axis. The 'bands' of pressure change can be clearly observed.

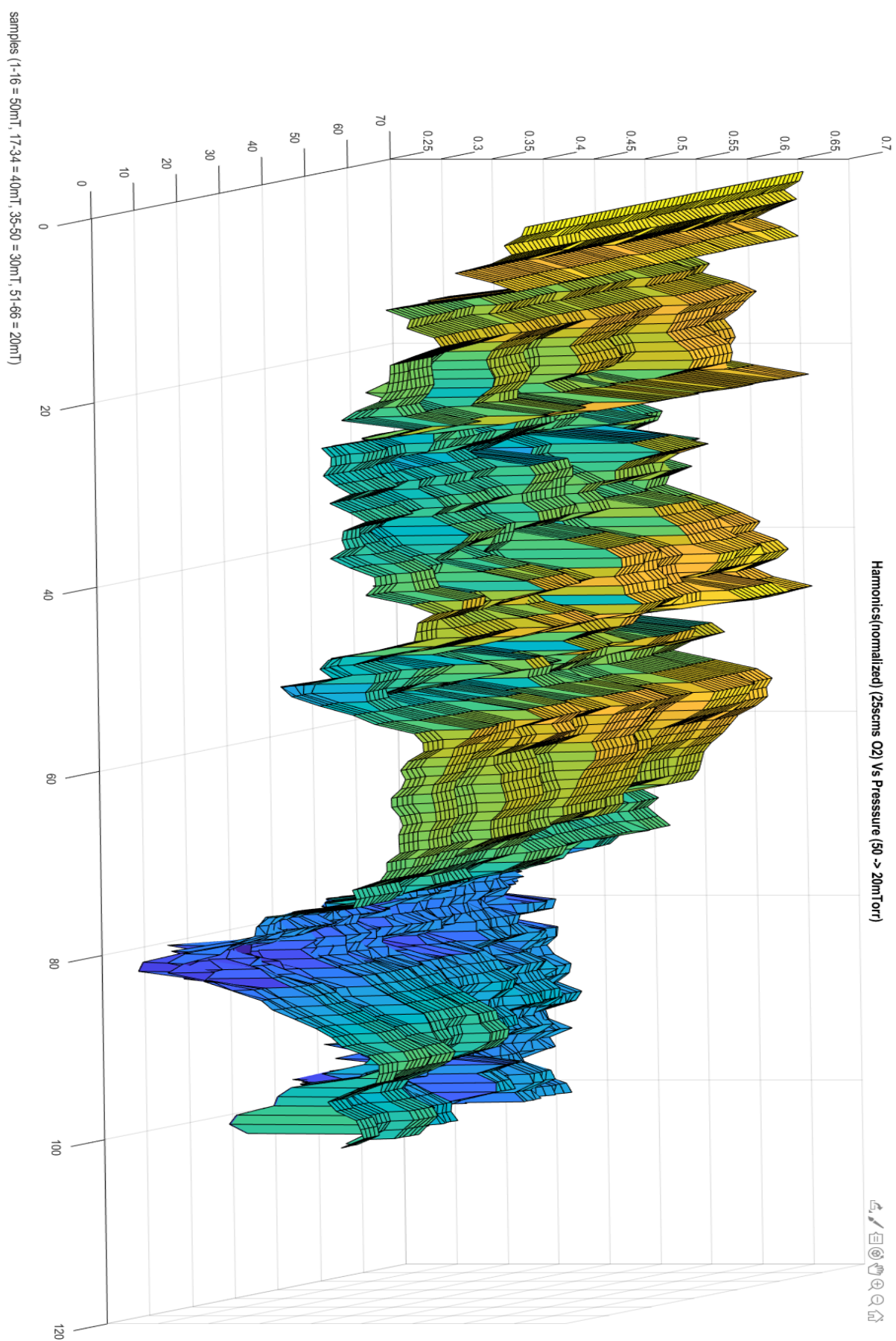
The graph legends are as follows:

- X = Harmonic number
- Y = normalized power of harmonic
- Z = samples (number)
 - 1-16 for 50 millitorr,
 - 17-34 for 40 millitorr
 - 35-50 for 30 millitorr and
 - 51-66 for 20 millitorr

The three planes are then plotted for the different flow rates of O_2 .

Hamonics(normalized) (20scms O2) Vs Pressure (50 -> 20mTorr)





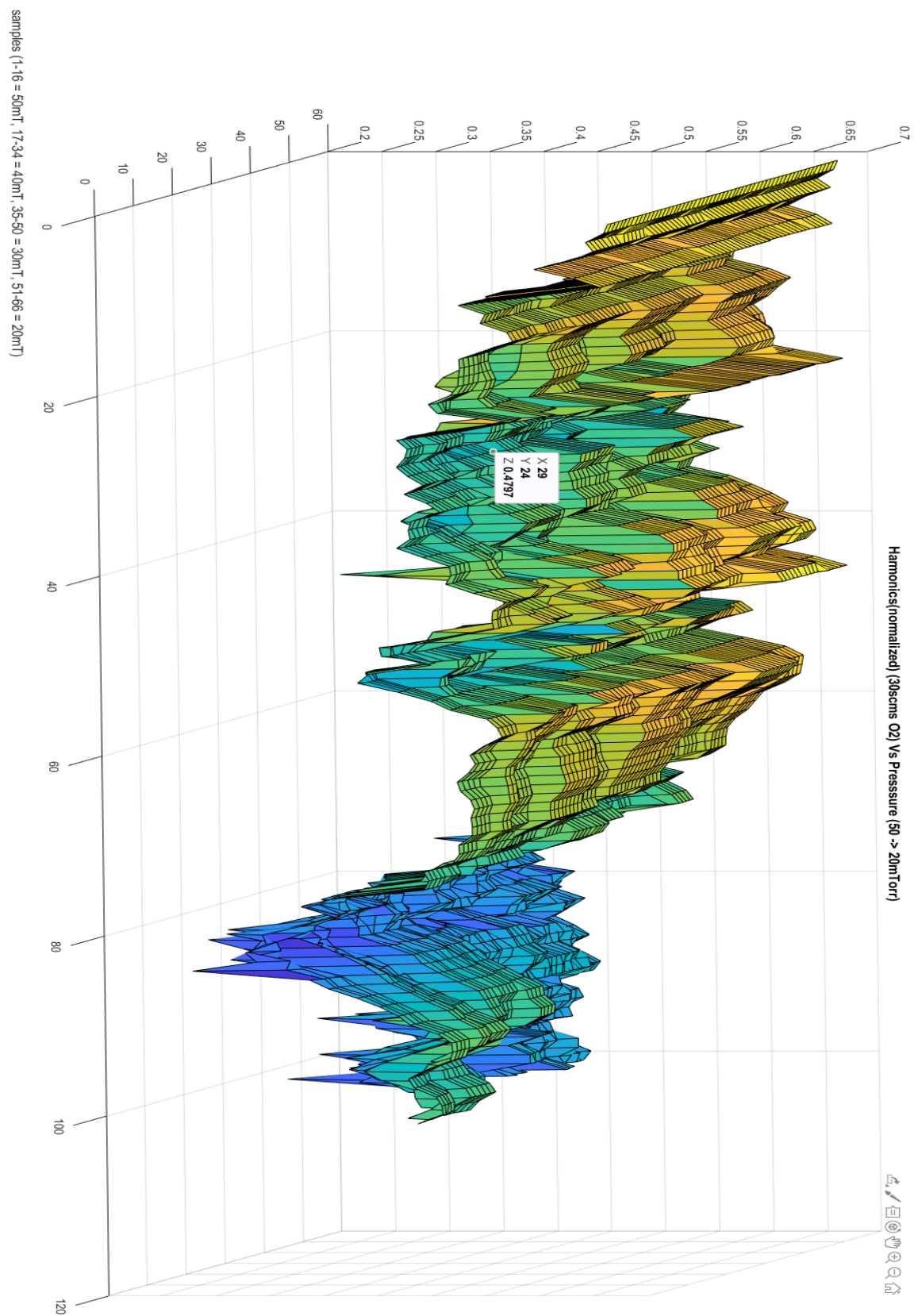
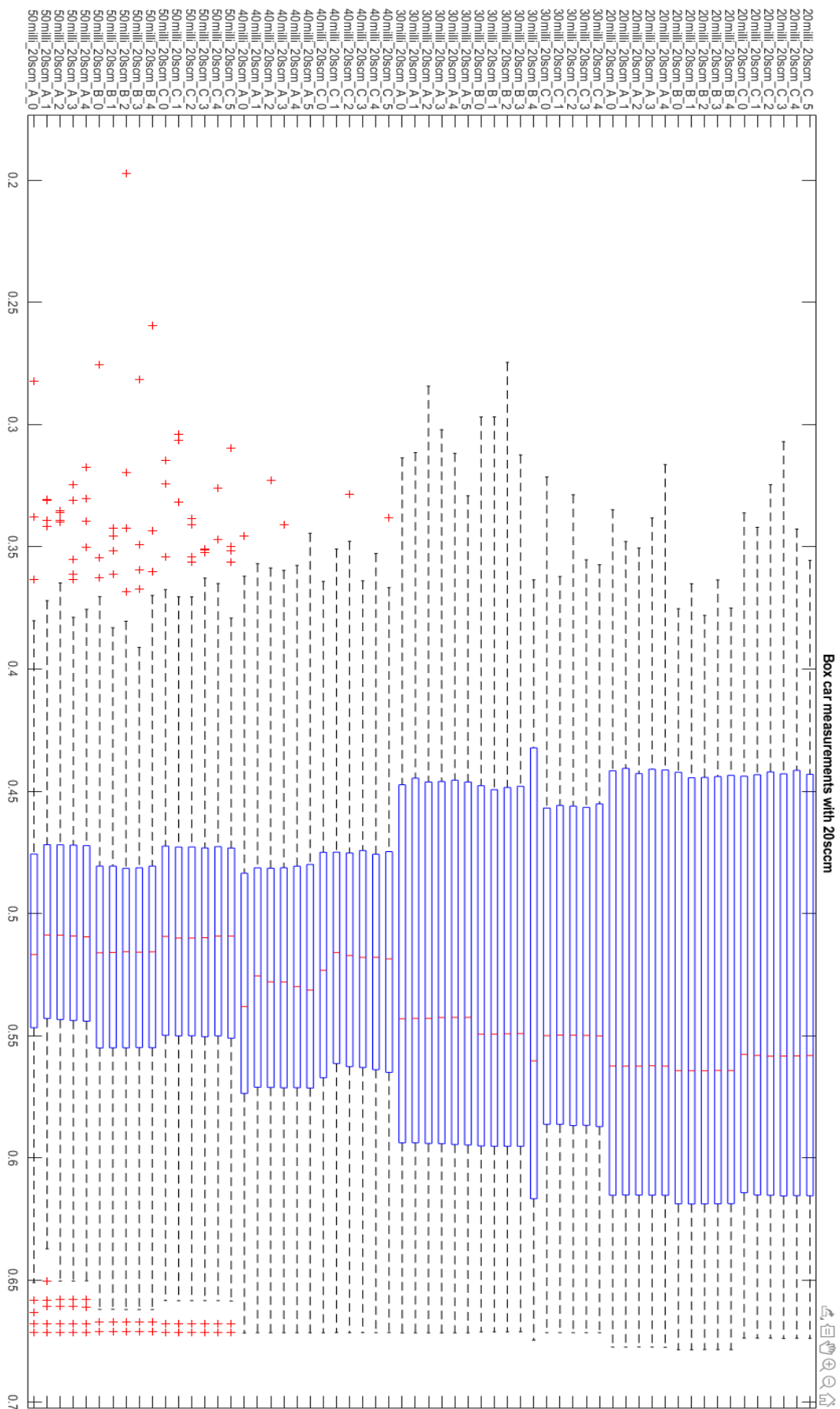
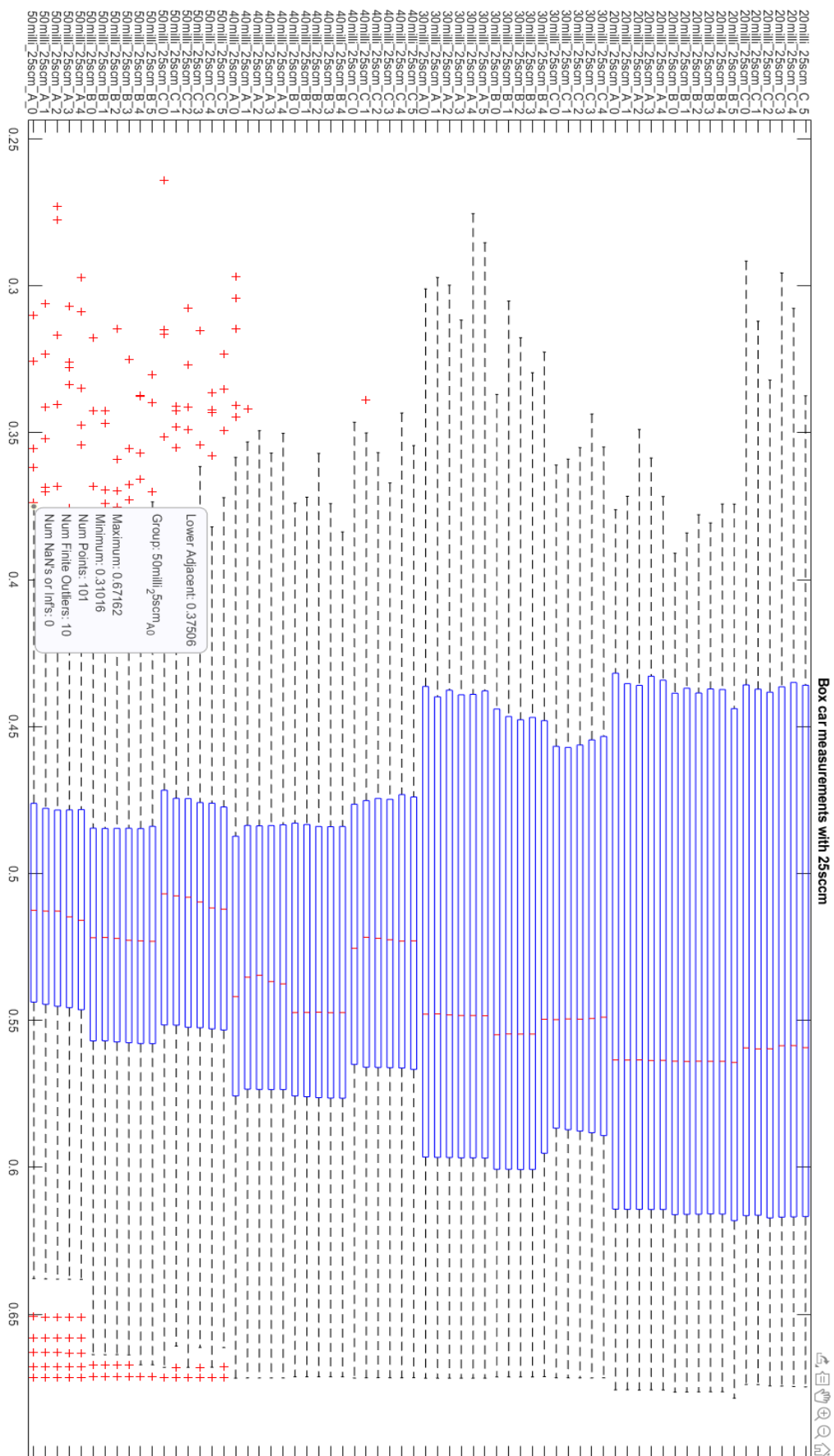


Figure 97 Harmonic Planes for 20, 25 and 30 sccm O₂, respectively

From the figures above, we see that the recorded harmonics take on a similar form along the X- and Y- axis (left to right and vertically), but the pressure changes can be clearly distinguished as shown in the Z-plane (into the graph) where the distinct changes can be observed in the recorded values.

By examining the boxcar plots for each of the harmonic recordings for each pressure and gas-flow regime, we can evaluate the variance and the ranges for the various harmonics as the pressure is varied. One immediate observation is that there are changes in median and quartile points for no changes in 'steady-state' plasma parameters. This is an indication of systematic errors in the recording of the harmonics. Further work is required to make more exact recordings. On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the '+' symbol.





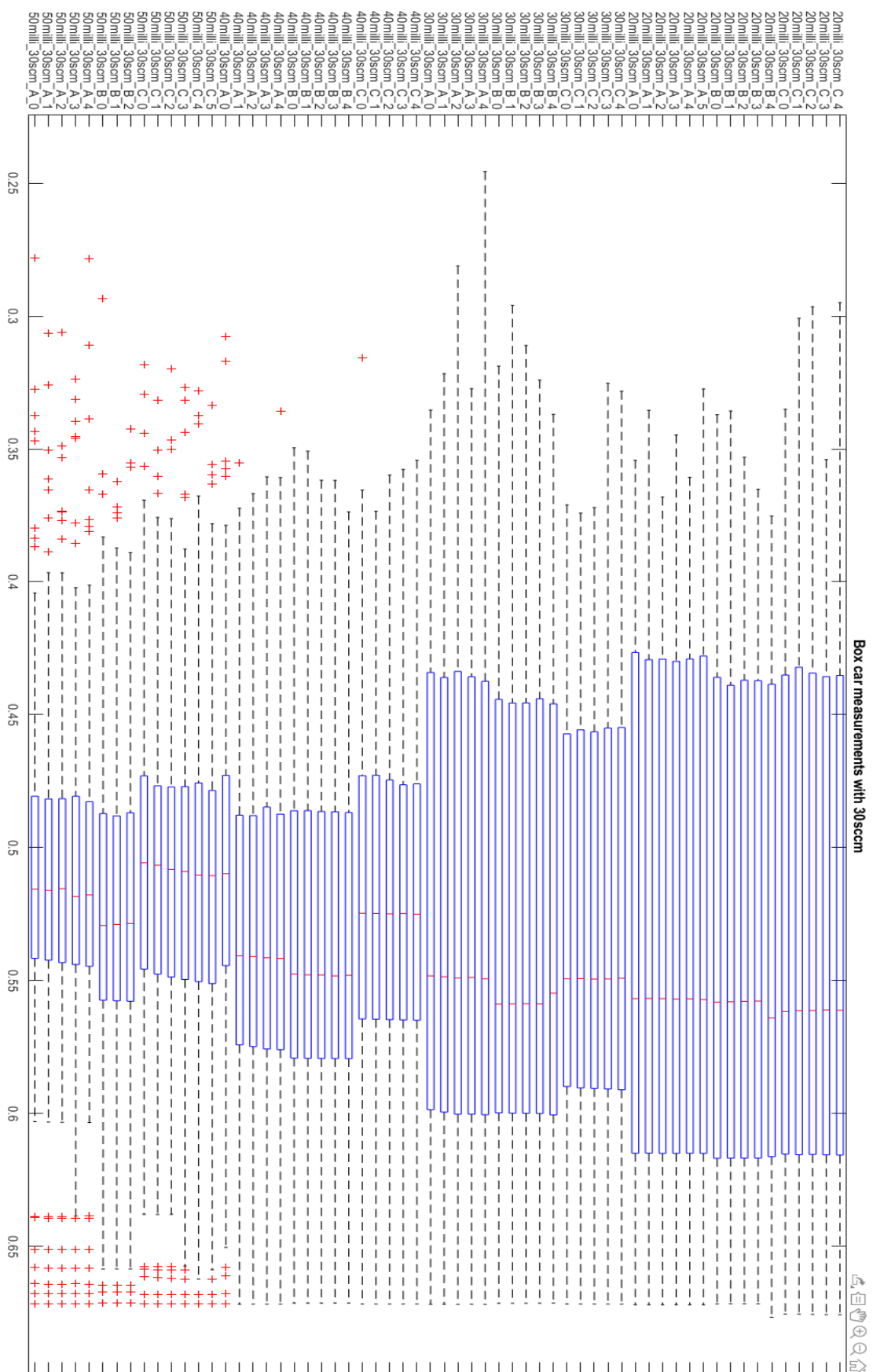
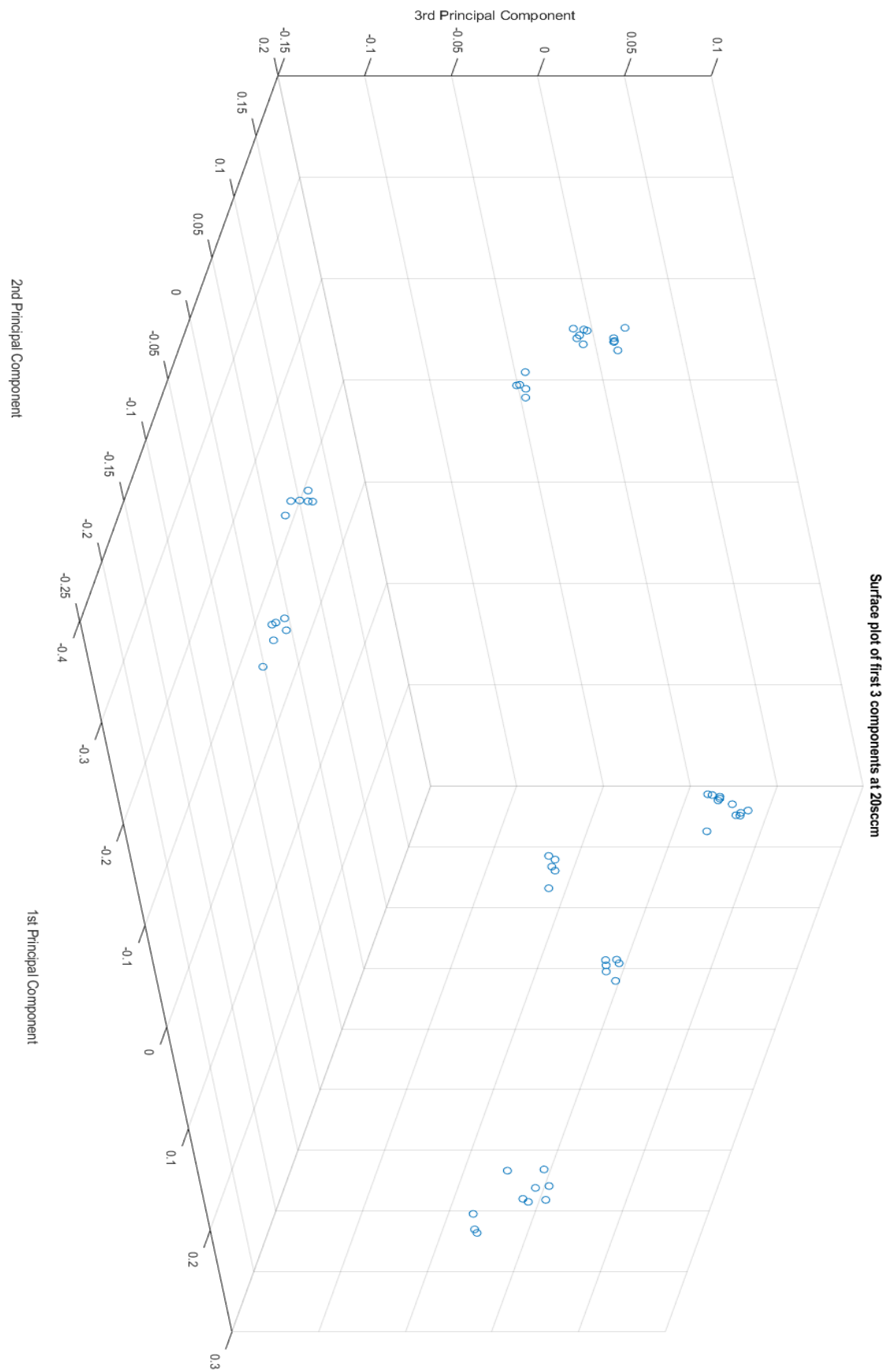
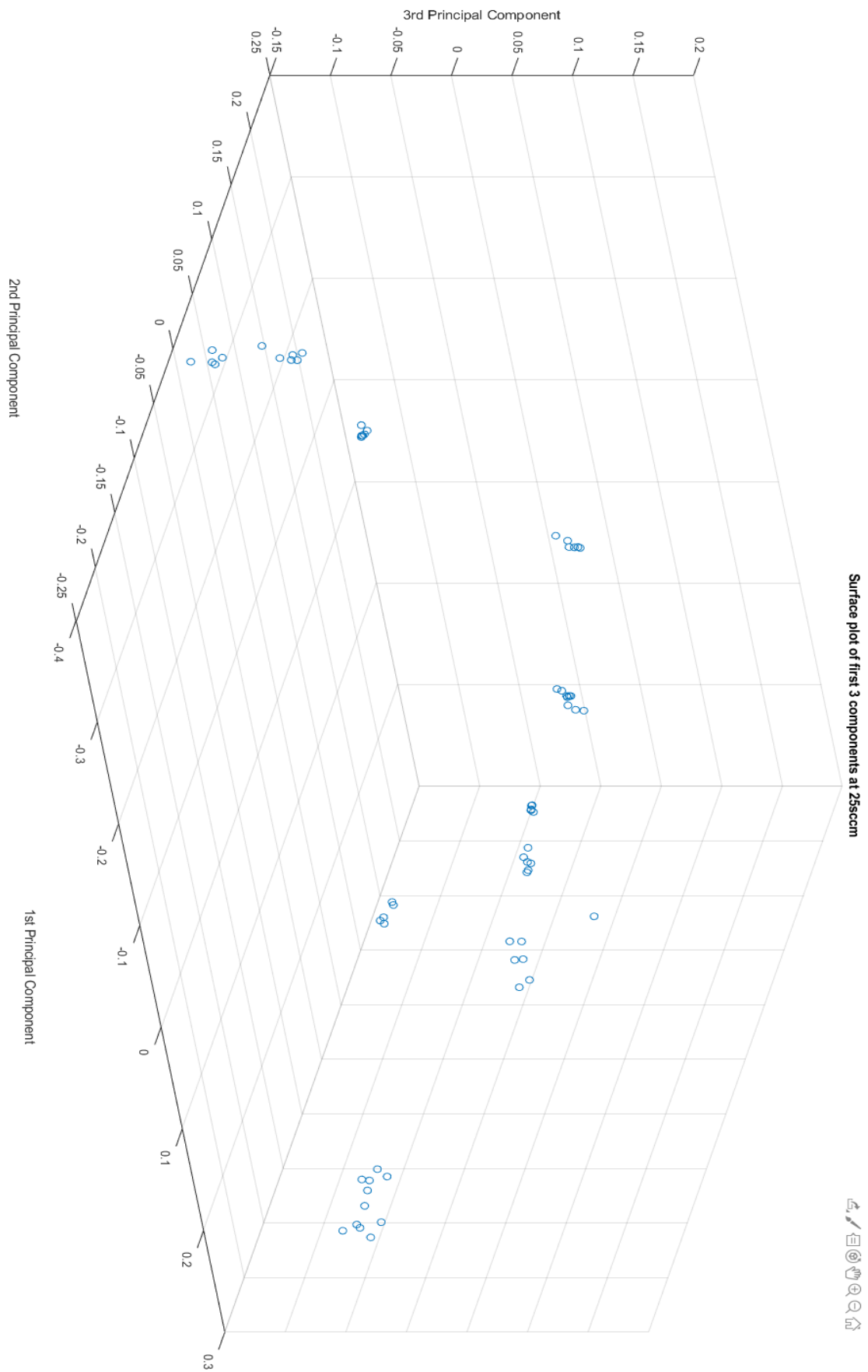


Figure 98 Box Car Plots for the Three Gas-flow Regimes of 20, 25 and 30 sccm, respectively

By visualizing the first three principal components we can see the clustering of the measurements into distinct sets. The principal component analysis will be shown later. By visualizing the data, it is possible to determine the analysis method most suitable for the KNN model. The principal component may be used as a reduction method for KNN classification, but in this the design we use the full complement of harmonics and some derived parameters to describe the hyperspace more fully. No dynamic classification has been performed as the sample rate of the initial acquisition was too slow.

By plotting a 3D space representing the first three principal components, we can observe distinct clustering of the harmonic data. We will see later that the first 3 principal components account for 97.4% of the determined variation in harmonic content.





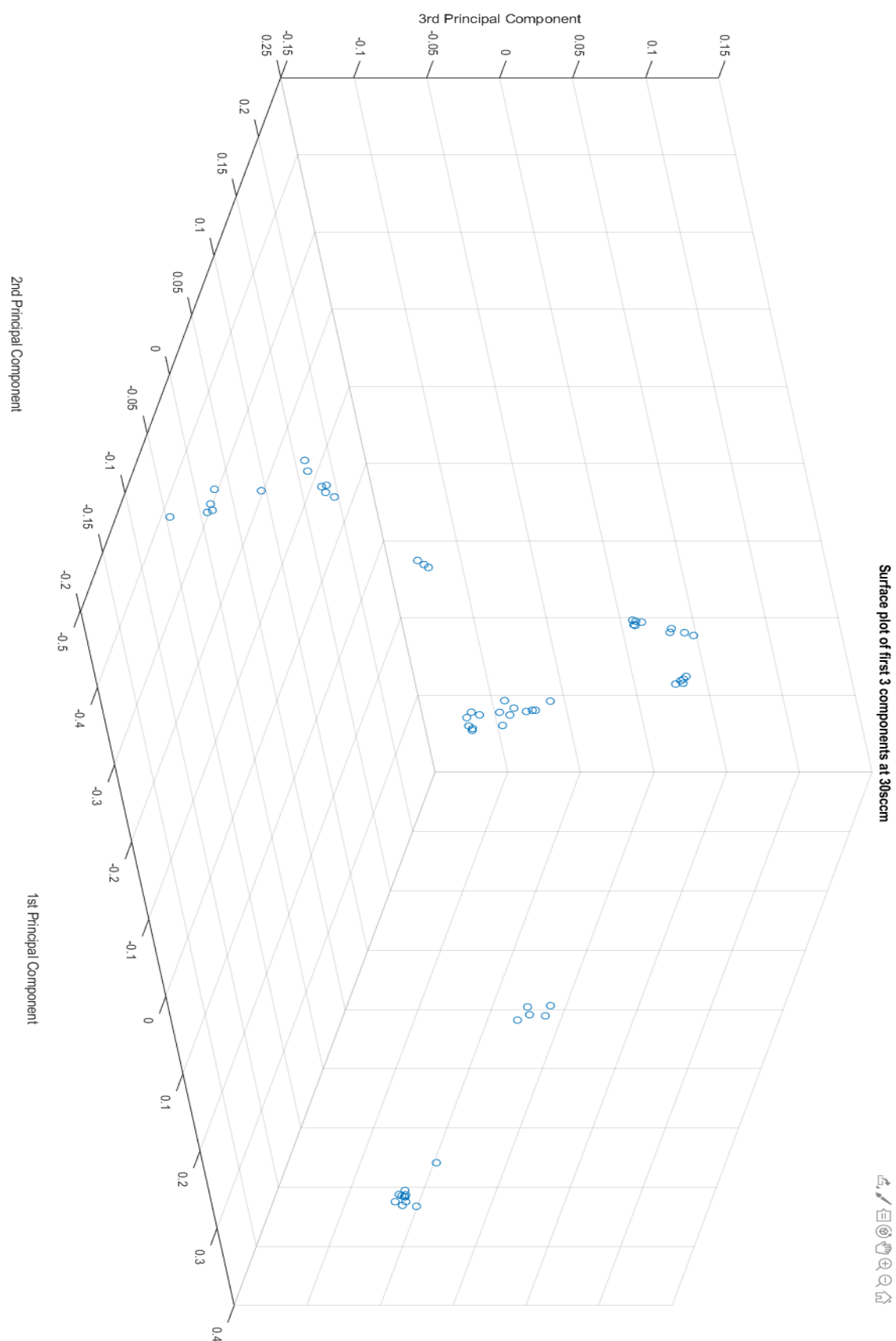


Figure 99 Comparison of First Three Principal Component Loadings for 20, 25, 30 sccm Gas-flows, respectively

Whilst the general form of the clustering is the same for the three gas-flow regimes, the models (shown later) utilize the pressure as a precursor for predicting the gas-flow.

6.4.3. KNN for Pressure (Single Linear Mean Model)

The FRES harmonic measurements were all tagged and one third (1/3) of them were used for calculating the averages to produce the centroids. The means of the harmonics were calculated and these were used to generate the centroids for the linear pressure KNN model. Then the complete set of remaining tagged FRES harmonic vectors are input to the model and classified using this KNN (101 dimension) hyperspace. The following results are obtained.

For the tagged inputs of 'unknown' data the following accuracy tracking can be observed:

Linear KNN Pressure Model	Pressure Accuracy	Gas Flow Accuracy
	190/192(samples) = 98.95%	127/192 (samples) = 66.14%

Table 6 KNN Linear Means Accuracy for Pressure and Gas Flow

Initial pressure prediction results are very promising. Since the tracking accuracy for gas flow was less accurate, this is investigated further. The gas flow prediction method was divided into segmented models of known pressure. This will be shown in the following section.

6.4.4. Segmented KNN for Gas-Flow Rates

The above experiment utilized a single mean hyperspace generation for the entire data set of Pressure vs. Gas-Flow flow settings. To determine if there was a correlation between gas-flow rates and the harmonic content, separate models were developed for the constant pressures. Four KNN models were developed for the static pressures of 20, 30, 40 and 50 millitorr, respectively. These models were trained using one third (1/3) of the measurements to produce means. These means were used as the centroids for the four KNN models for each of the known pressures of 20, 30, 40, and 50 millitorr.

For a pressure of 20 millitorr, a KNN model based on the means of the measurements was generated. This was repeated for the other pressures, generating four models. With a high accuracy of prediction for pressure, the specific prediction model was used to predict the gas-flow. The following results were obtained:

Linear KNN Gas Flow Model	Gas Flow Accuracy
20 millitorr	36/49(samples) = 73.47%
30 millitorr	36/47(samples) = 76.59%
40 millitorr	25/49(samples) = 51.0%
50 millitorr	30/47 (samples) = 65.22%

Figure 100 KNN Model Prediction Accuracy for Gas-flow Rates

Even accounting for this 'known pressure' and building a model for the gas-flow rate, the results were not as expected. A change of the flow-rate of O_2 should see a corresponding change in the chemistry and subsequent harmonics of the plasma. It was considered that higher speed acquisition may permit the observation of jerk in the harmonics and the KNN models may be extended with kurtosis and jerk added to the hyperspace dimensions to aid in the identification of gas-flow rate changes.

To further examine the gas-flow rates at different pressures, the harmonic variances for gas-flow changes were plotted for the different pressure regimes. This is shown in the figure below (see Figure 101).

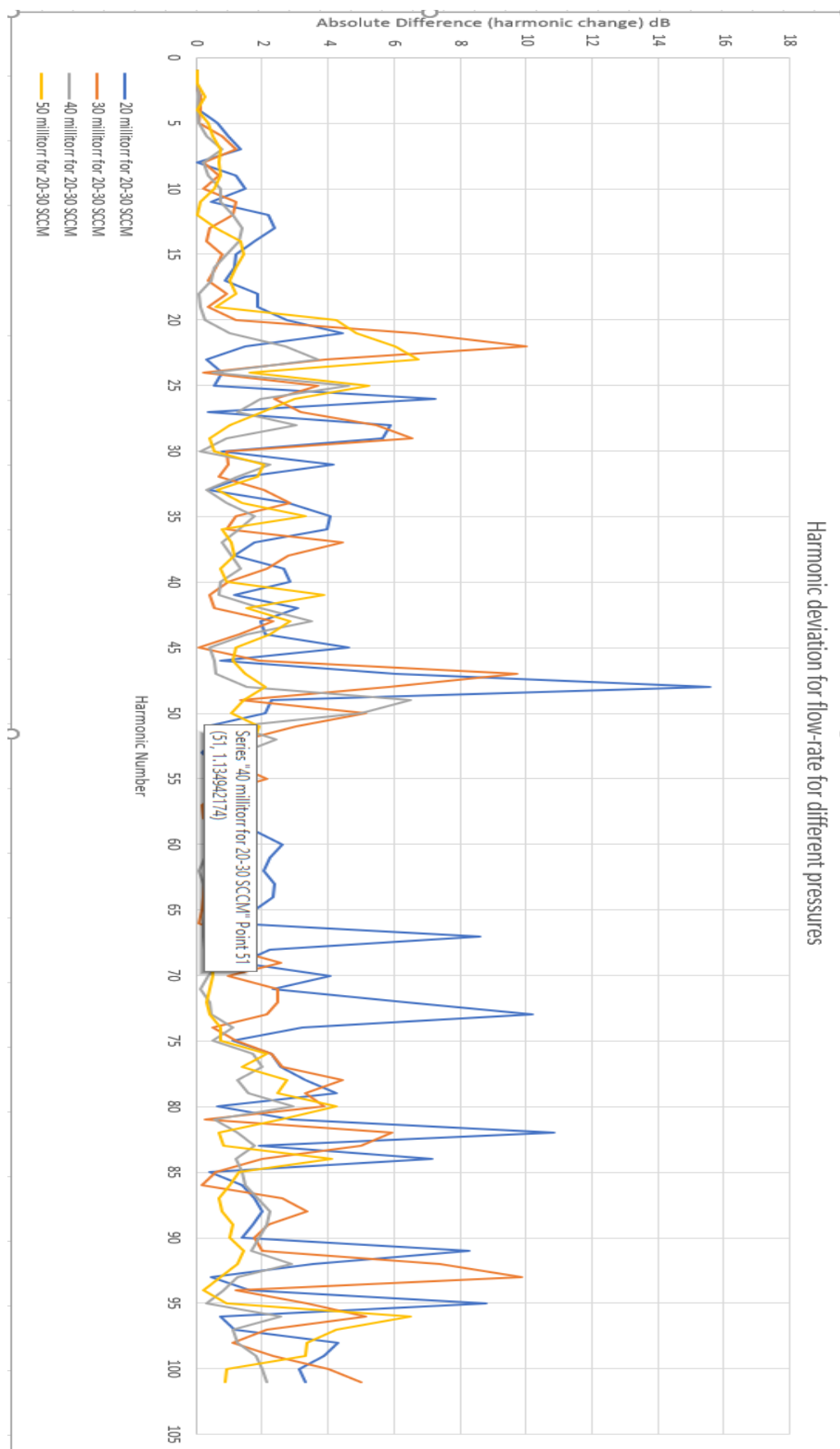


Figure 101 Variation of Harmonics as a Function of Gas-flow for Different Pressures

The variance of the harmonics was examined to see if specific frequencies contributed more to the principal component. Of note was the varied harmonic variance for the different pressures. It can be seen that gas-flow changes affect different harmonics at different pressures. For example, at 20 millitorr pressure, harmonic 21 signifies a change in gas-flow with a difference of 4 dB, at 30 millitorr, harmonic 22 has a change of 10 dB for the gas flow change, at 40 millitorr harmonic 23 has 3.6 dB change and finally at 50 millitorr, harmonic 23 has a 6.7 dB change when the gas flow rate is changed from 20 to 25 to 30 sccm.

This change of harmonic contributing to the identification of the gas-flow rate can be more readily observed when the loading matrix of the PCA is plotted for each gas-flow rate. The differing peaks and troughs indicate the correlation (strongly positive or strongly negative) for the gas-flow rates for the different pressure regimes.

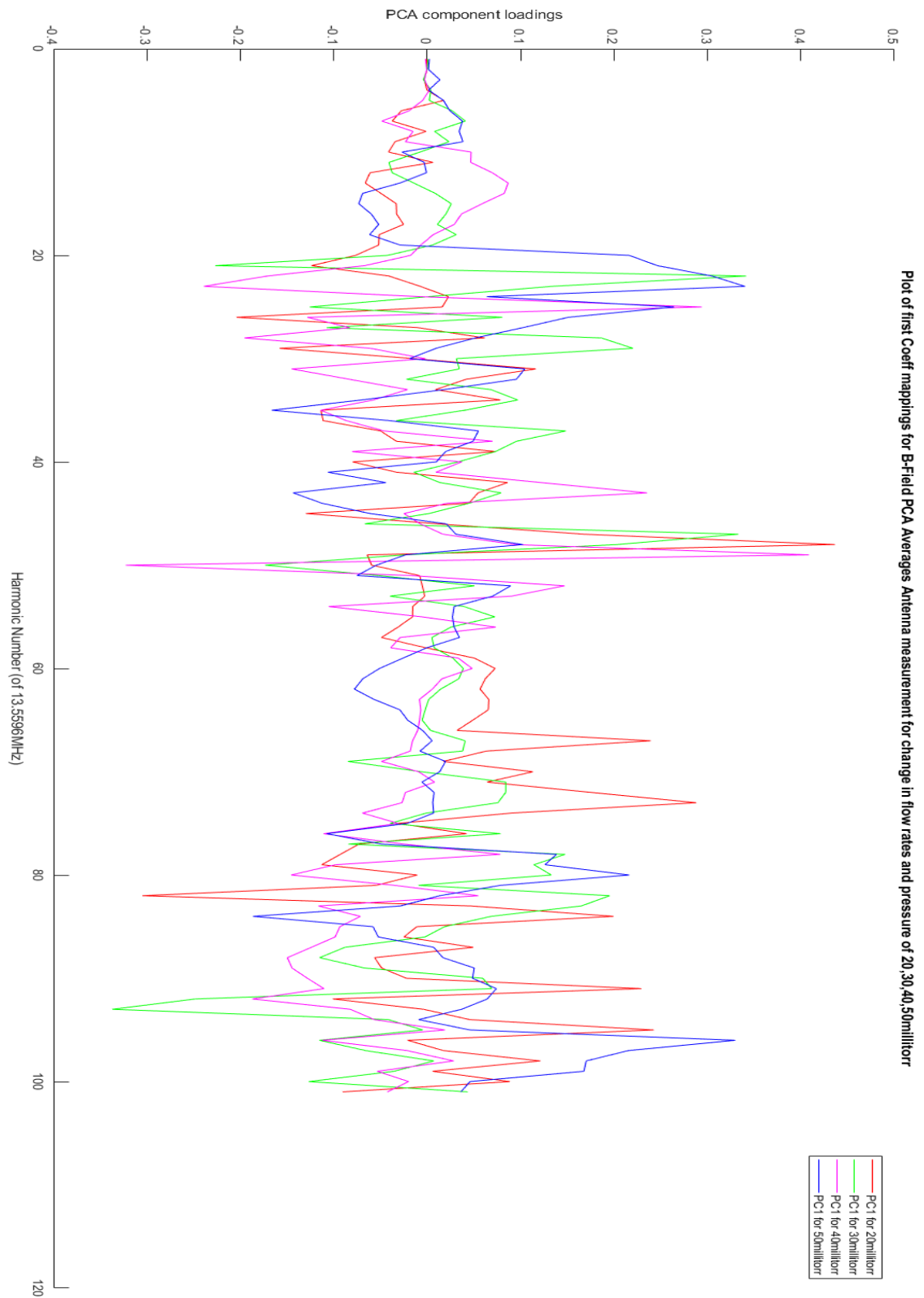


Figure 102 First Principal Coefficient Variance of Harmonics for 20, 30, 40, and 50 millitorr Changes

The harmonic variation for the different gas-flow rates occurred at different harmonics when different pressures were in place. It can be observed, that for changes in gas-flow rates, different harmonics varied depending upon the absolute pressure within the chamber. (This may be an interesting area of further research).

6.4.5. Boosted KNN for gas flow prediction at different pressures

The inclusion of all the harmonics to produce centroids generates unnecessary noise to the prediction classification. This is due to the equal weighting of the harmonics in a Euclidean distance measurement. For different pressure regimes, it is necessary to locate only those harmonics that contribute to the classification of the gas-flow. It can be observed from the figure (Figure 102) above, that a particular pressure determines the weights to be applied to the harmonics for extracting the gas-flow rates. In the next section, this will be more closely examined in principal component analysis and it is shown that the pressure regime changes the salient harmonic indicating the change in gas-flow rates. The measurements were taken over a number of days with 3 experimental regimes, these being experiments A, B and C. For matching static plasma parameters, the measurements were compared to see how well they matched up. Two particular error streams appear. One of these error-streams has been observed in the box-car measurements where they are seen as outliers. By evaluating the data sets for equivalent operating points (e.g 20 millitorr and 20 sccm, 25 sccm and 30 sccm), the following absolute variations were observed. These are the plotted variations for the 3 experimental regimes and show very large deviations.

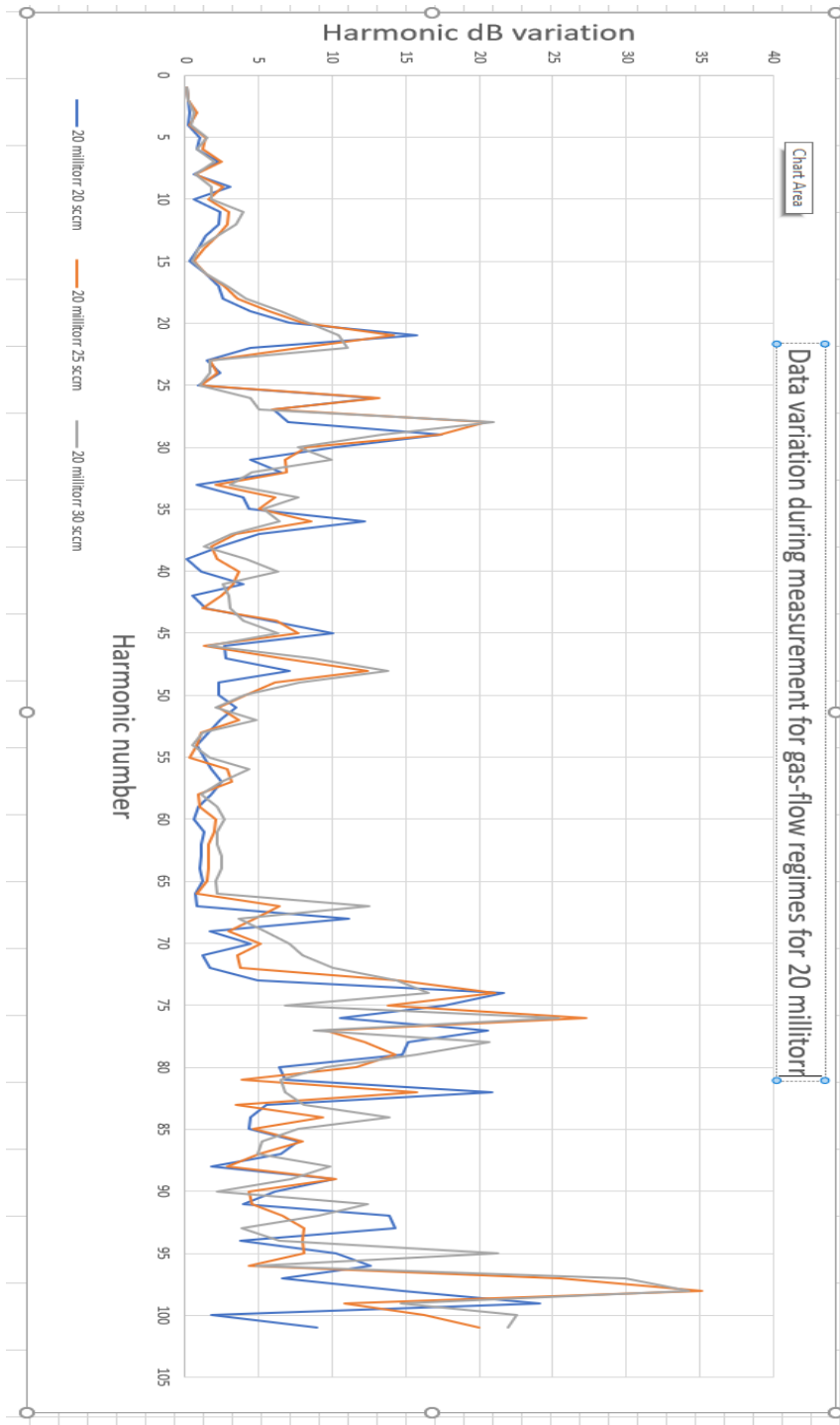


Figure 103 Experimental Measurement Variation for 20 millitorr Chamber Pressure.

In the multi-experiment acquisition of the data, up to a 37 dB variation in harmonic measurement is observed and this is for the system operating in steady state of 20 millitorr, but observed for different days. Similar variation graphs for

the same steady-state plasma conditions show this variation over the experiments.

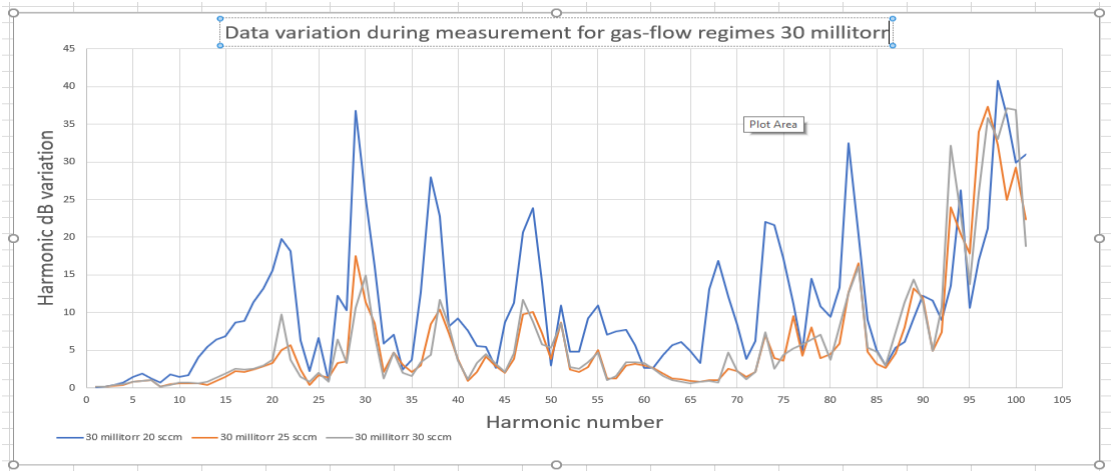


Figure 104 Experimental Measurement Variation for 30 millitorr Chamber Pressure

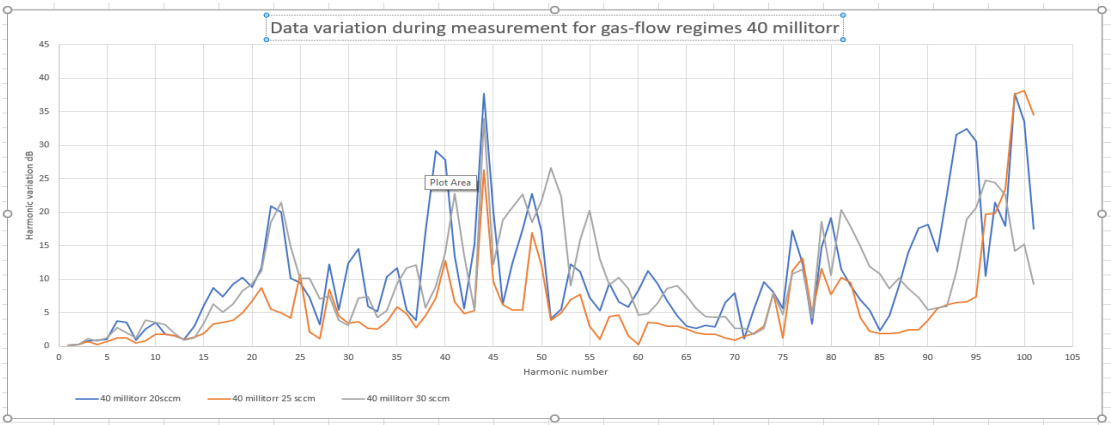


Figure 105 Experimental Measurement Variation for 40 millitorr Chamber Pressure

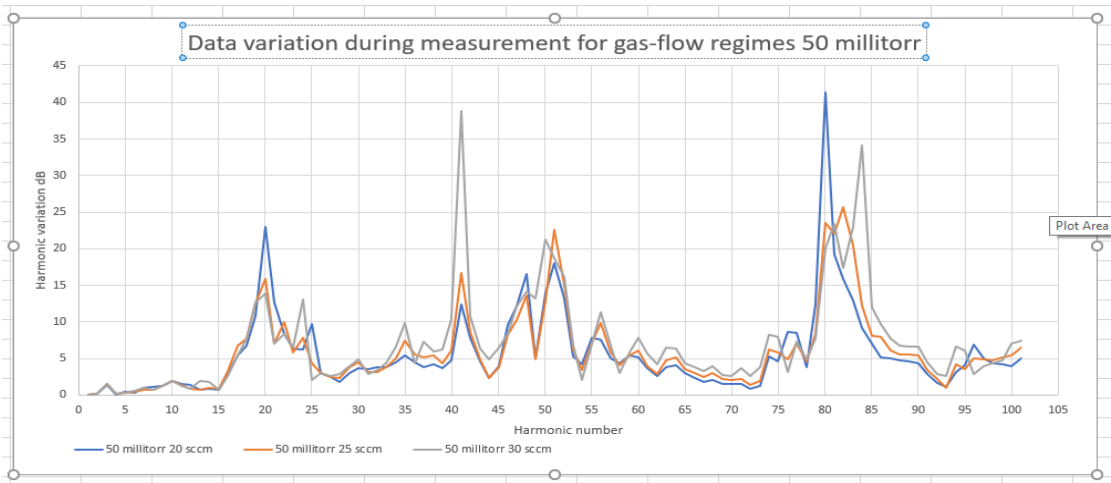


Figure 106 Experimental Measurement Variation for 50 millitorr Chamber Pressure

Comparing the differing experimental regimes of A, B, and C, it was determined that the actual values within any one particular experiment take on a steady state value (bar the outliers). We use this additional knowledge to build the centroids of the KNN model. By taking the individual means of the 3 experimental regimes for one third ($1/3$) of the total points, we build a new model that takes into account the experimental measurement variation. This gives a hyperspace that is 3 times larger now covering the experimental variation across the 3 experimental regimes A, B and C. This gives the following results.

Linear KNN Gas Flow Model	Gas Flow Accuracy
20 millitorr	47/49(samples) = 95.92%
30 millitorr	43/47(samples) = 91.49%
40 millitorr	39/49(samples) = 79.59%
50 millitorr	45/47 (samples) = 95.74%

Figure 107 KNN Model Prediction Accuracy for Gas-flow Rates Accounting for Experimental Measurement Variation

This indicates that gas flow rate can be determined with a high degree of confidence based upon the measurement of the harmonics using FRES. The variation in the data can be explained by the different recordings made of the harmonics over the different experimental regimes A, B and C. The location of the cabling, antenna and the use of the particular spectrum analyzer and balun system is of vital importance for repeatable measurements. To mitigate this on a

production tool, a 'step recipe' can be run at a regular interval with known gas flow rates and pressures to calibrate the antenna acquisition system to generate a known model for the KNN. (This model could be further analyzed for drift to see if the calibration procedure is performing as expected). This will remove the variation in measurements encountered in the manual application of data acquisition during the experimental stage. The 40 millitorr regime has the most significant experimental variation across the harmonics with a maximum value nearing 37 dB. Accurate and repeatable antennae placement and concise measurements should remove these inconsistencies. The appendix (See Appendix R) contains the code for the calculation of the staged KNN model to build a robust and accurate model for prediction.

6.5. Principal Component Analysis of Pressure (and Gas-Flow)

For the measurements of pressures 20, 30, 40, 50 millitorr and gas-flow rates of O_2 at 20, 25 and 30 sccm, the principal component loading matrix was calculated for the harmonic recordings. This is chiefly used in component reduction, by finding the principal variances, but 'loading' vectors indicate those harmonics that may optimally be recorded and measured for accurate determination of the metric under examination (see section below). The following measurements were determined:

Principal Mapping Vector	Principal Component Mappings Account
1	Accounts for 78.15742% of the variance
2	Accounts for 12.71379% of the variance
3	Accounts for 6.55560% of the variance
4	Accounts for 1.19071% of the variance
5	Accounts for 0.62416% of the variance

Table 7 Principal Components and Contributions for Harmonic Measurement Optimization

This indicates that a single vector accounts for 78% of the variation and that the top three of the principal component mappings account for most of the observed variation in the harmonic content for pressure-changes and the gas-flow changes.

This is plotted below:

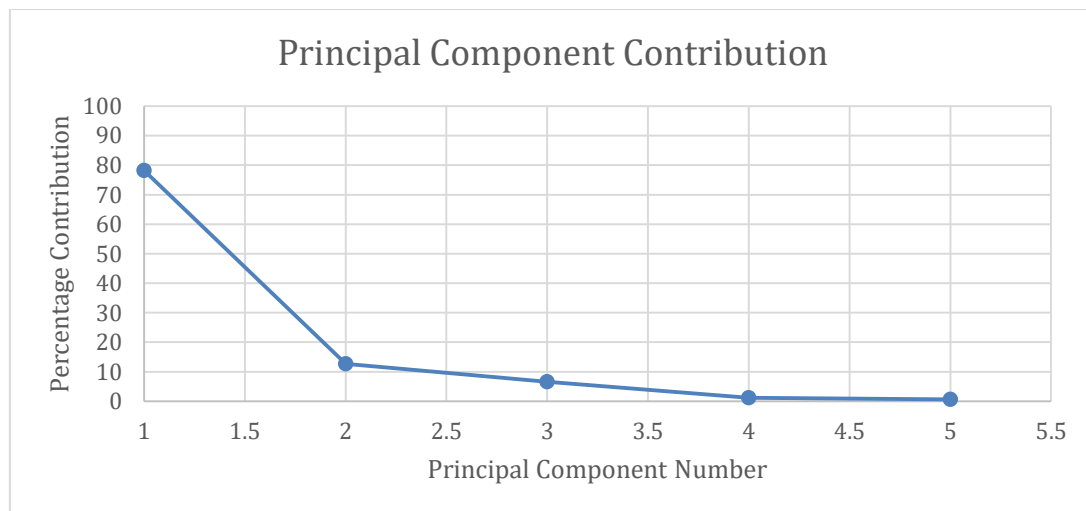


Figure 108 Harmonic Principal Component Contribution for Pressure and Gas-flow Measurements

The principal component vectors describing the planes of significant contribution are plotted below:

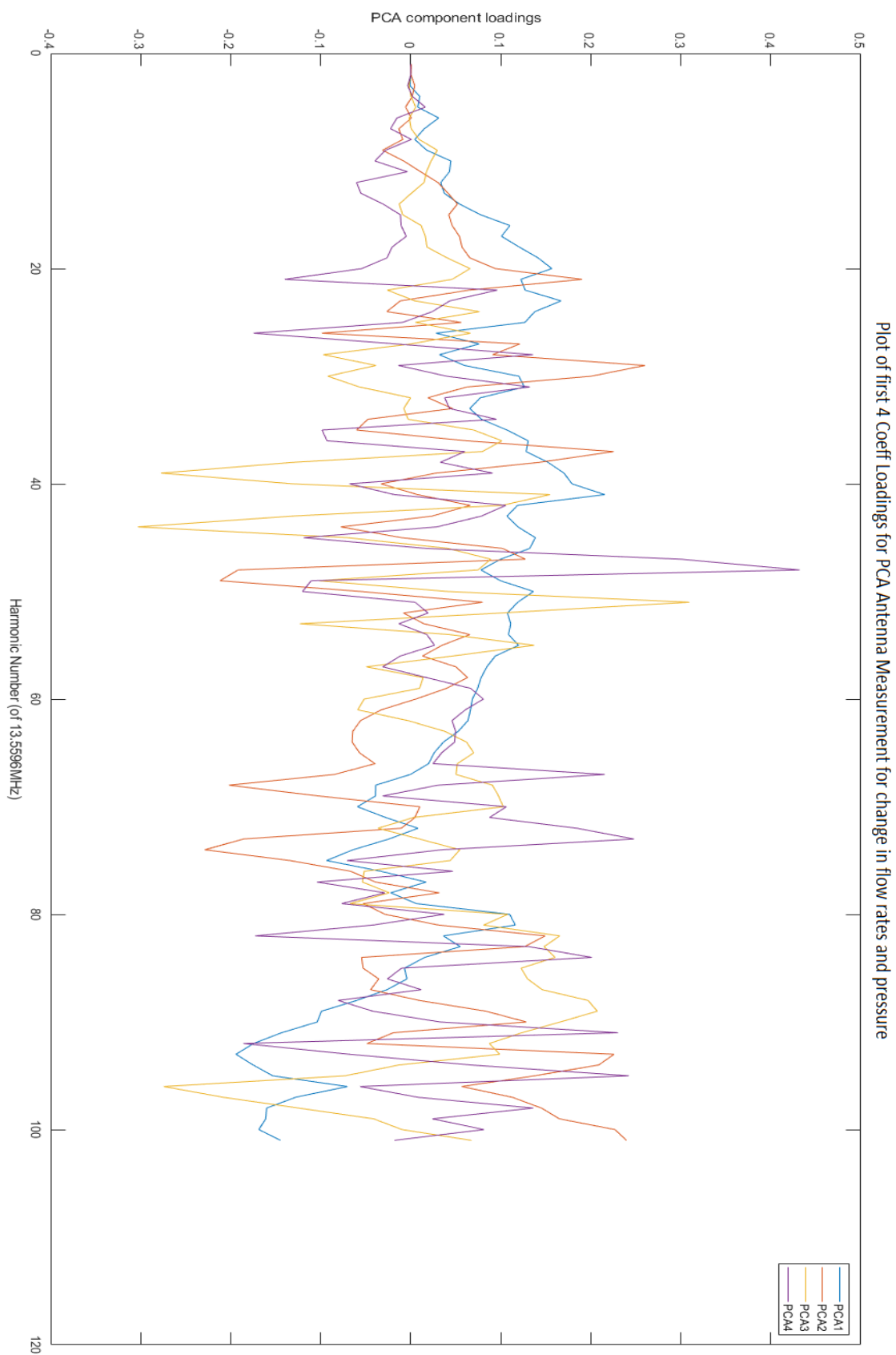


Figure 109 First Four Principal Component Planes for Plasma Pressure and Gas-flow FRES Measurements on Oxford Instruments PlasmaLab100 CCP Tool

The variances of the measured harmonics indicate those that are sensitive to the pressure and gas-flow changes. A large positive value indicates positive correlation between the harmonic and a large negative value indicates the contrary. The harmonic variance shows the harmonic sensitivity to the parameter changes that have been applied to the plasma in the chamber. The larger variances indicate the priority harmonics to be recorded for the measurement (in this case) of the pressure and the gas-flow. From a product point of view, this may be dynamically calculated and the optimization is to use the 'sweeping' resources of the radio acquisition system to concentrate on those harmonics giving the most variation.

6.6. Harmonic Recording Optimization

The above experiments recorded 101 harmonics of the fundamental drive frequency (13.5596 MHz) of the Oxford Instruments PlasmaLab 100 and this entailed a large hyperspace to describe the space. Further analysis of the data permits an optimization of the recordings required and the ancillary steps associated with the acquiring, processing and storage of these harmonics. By performing a Principal Component Analysis (PCA) on the data for the tagged items, we can derive an optimal set to dynamically select those harmonics that are required to be acquired and processed. Given the data set for the Pressure vs Gas-flow measurements that were made, the PCA is performed and the variance is plotted.

The level of the variance determines those harmonics that should be prioritized over the other ones.

6.7. Summary

The novel use of the KNN modelling technique using the FRES analysis (and TRES acquisition and processing) is a very powerful tool in the control and monitoring of an industrial plasma system. With accurate measurement of the FRES (and TRES processing), very concise feedback control can be applied to an operating plasma system. The PCA analysis for any particular operating mode can be used to isolate those harmonics that contribute specifically to the measurement under observation. It is not fully understood why particular harmonics are sensitive to particular operating modes such as gas-flow and pressure, but having a method to determine those sensitive sets is of particular use in an industrial environment for the closed-loop control of the plasma process. The acquisition can include all of the harmonics, but subsets can be extracted to be fed to the models in order to accurately predict the operating points and this may be used in a closed-loop feedback system. The independent predictions from the multiple KNN models may be used to feed a voting system to close the loop on the operation on the plasma system.

This page intentionally blank

Chapter 7 Laplace Transform Modelling of Oxford Instruments PlasmaLab 100 CCP

7.1. Introduction

We have seen in the FRES system, that the E-field and the B-field RES signals can be decomposed into their constituent harmonics. Whilst the Fourier transform is a method to gauge the combination of sinusoids in a signal, the Laplace transform is a method to determine the sinusoids and the exponentials in a signal. This transform plots the Fourier in the 2D complex dimension. Thus, the Fourier transform can be viewed as a 'slice' of the Laplace transform. We use this to examine the operation of the plasma as a set of linear differential equations and map the components to the continuous time equations governing the flows within the plasma.

We have seen

$$\text{Fourier Transform } X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t} dt$$

For a given function of time this extracts the constituent sinusoids within the waveform, together with their relative weighting functions. We note that for $\omega = 0$, the exponential term goes to 1 and the integral is just the area under the curve of the time signal (which indicates the dc bias component).

$$\text{Laplace Transform } X(s) = \int_0^{\infty} x(t)e^{-st} dt$$

where we are dealing with real signals, which are not defined for $t < 0$

and where

$$s = \alpha + i\omega$$

and s is comprised of a real component (α) and an imaginary component ($i\omega$).

Thus, the Laplace transform of $x(t)$ is the Fourier transform of $x(t) e^{-\alpha t}$. We use the Laplace transform of the harmonics to generate a transfer function from the known input to the measured output.

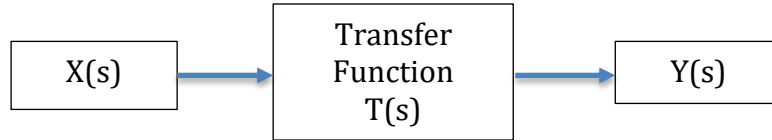


Figure 110 Transfer Function of Oxford Instruments PlasmaLab 100 CCP System

In the case of the Oxford Instruments PlasmaLab 100 CCP system

$X(s)$ is 13.56 MHz CW sinewave

$Y(s)$ represents the measured harmonics as recorded

$T(s)$ is the transfer function including the Matching Unit, the cables, the plasma, and the measuring antennae subsystem.

7.2. Basic Transfer Function

We model the basic system as seen above (Figure 110) and one can see that the transfer function is

$$T(s) = \frac{Y(s)}{X(s)}$$

where $X(s) = \int_0^\infty \sin(\omega t) e^{-st} dt$

and for $\sin \omega t$, $\omega = 2\pi f$, $f = 13.56$ MHz

This can be simplified to

$$X(s) = \frac{s}{s^2 + \omega^2}$$

which has poles at $-j\omega$ and $+j\omega$ and a zero at the origin. This would correspond to a Fourier transform with two poles either side of the axis representing the frequency of the primary drive and no other frequencies. From the measurements of harmonics taken by recording the B-field of the plasma at the viewport on the Oxford Instruments PlasmaLab 100 CCP, the output of the plasma is a complex set of harmonics that vary with the plasma parameters. For these experiments, we recorded the first 103 harmonics (primary drive not recorded) of the fundamental drive sinewave of 13.5596 MHz. Using linearity of the Laplace transform,

$$L(a, f) = aF, \quad \text{and} \quad L(f + g) = F + G$$

we can use superposition to describe the measured waveform as a linear combination of sinewaves and get an approximation of the Laplace transfer-function of the plasma. In the steady state operation of an Ar plasma we have the following

$$Y(s) = a \cdot \frac{s}{s^2 + \omega_1^2} + b \cdot \frac{s}{s^2 + \omega_2^2} + c \cdot \frac{s}{s^2 + \omega_3^2} + d \cdot \frac{s}{s^2 + \omega_4^2} \dots\dots \text{(for 102 recorded samples)}$$

where a, b, c and d... are the linear magnitudes of the recorded harmonics, and ω_n represents the contribution of that n^{th} harmonic frequency.

The transfer function can be expressed in terms of the output function and the input function.

7.3. Oxford Instruments PlasmaLab100 Laplace Transfer Function

The following schematic model shows the layout of the measurement points and the relative scales of these values.

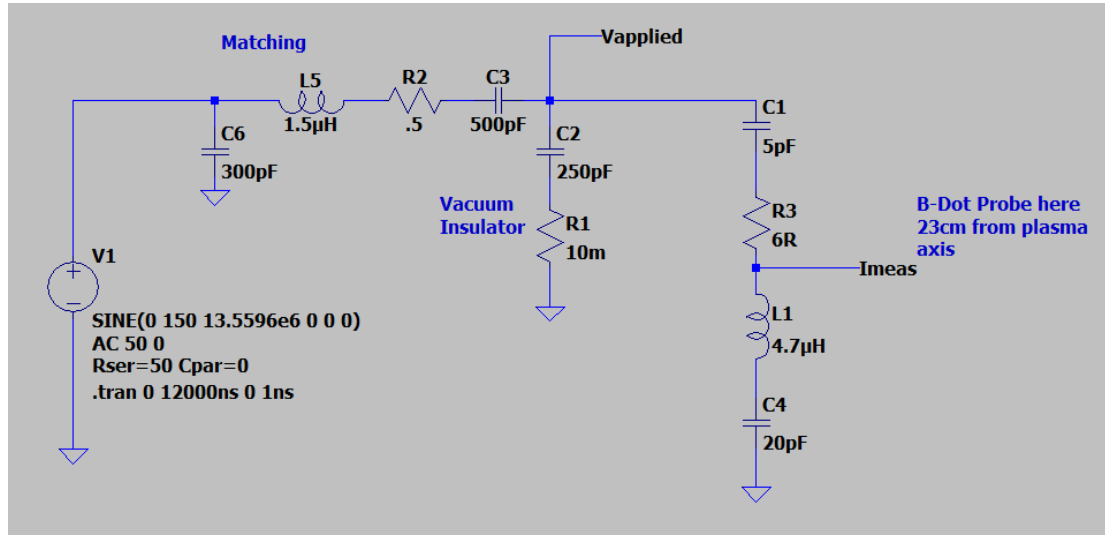


Figure 111 Basic Circuitry for Laplace Transform Modelling of Oxford Instruments PlasmaLab 100 CCP System

The values in the basic equivalent circuit are a combination of the capacitively coupled plasma system analysis by Schmidt *et. al.* (Frederik Schmidt J. T., 2019), from direct measurements of the Oxford Instruments PlasmaLab 100 CCP system and from the simulations performed in Chapter 5. We use the maximum values of the variable elements in the match unit, though in practice they will be trimmed by the auto-tuner to minimize the standing wave ratio (SWR) and maximize the power transfer from the RF power supply unit (PSU) to the plasma. The asymmetrical electrodes with sheath are modelled as C1 and C4 with values to represent the asymmetry of the electrodes. The bulk ohmic plasma is modelled with the series resistor (R3) and the inductor (L1). The matching unit and dc blocking capacitor are modelled using the combination of C6 and the series inductance of L5, R2 and C3. Cabling losses and insulator losses for the driver

electrode are modelled as C2 and R1. We set the simulated power-supply as a sine wave with an output of 150 V with a series resistance of 50 Ω and a sinusoidal frequency of 13.5596 MHz. We evaluate the transient simulation for 12,000 ns with a resolution of 1 ns (Figure 111) (.tran command line). The reference to the B-dot probe at 23cm is the B-field antenna located flush to the chamber wall and is 23 cm (as measured on the Oxford Instruments PlasmaLab 100 CCP system) from the axial centre of the bulk plasma between the active electrode and the grounded one.

The system is modelled as a combination of linear lumped elements and is simulated using the Spice range of analogue simulation tools.

The models (ML1) presented by Lieberman (Lieberman M. A., 1988) and the adiabatic model of Metze, Ernie and Oskam (MEO) (A. Metze, 1986) the modelled particles are inertialess, and are time-independent. Riley *et al.* (Riley, 1995) in presenting the Riley Unified Model (R.U.M.) (see Figure 112 below), attempts to address the inertia problem and the '*heavy ion limit*' by having a stepped function to control the sheath-ion current. This stepped function is a compromise situation where the total current is increased to 90% of the ion current. The Lieberman (ML1) model assumes that the ions respond to the full time-average of the RF cycle.

The ML1 model has no satisfactory dc limit and the MEO model only becomes accurate as the applied fields tend to dc. The Riley model (R.U.M.) combines features of both to give a 'compromise situation of 10% that gives reasonable results' (Riley, 1995). We present a method, such that by using the known input

and the measured output (the B-field at a single location), the direct transfer function of the operating plasma is obtained.

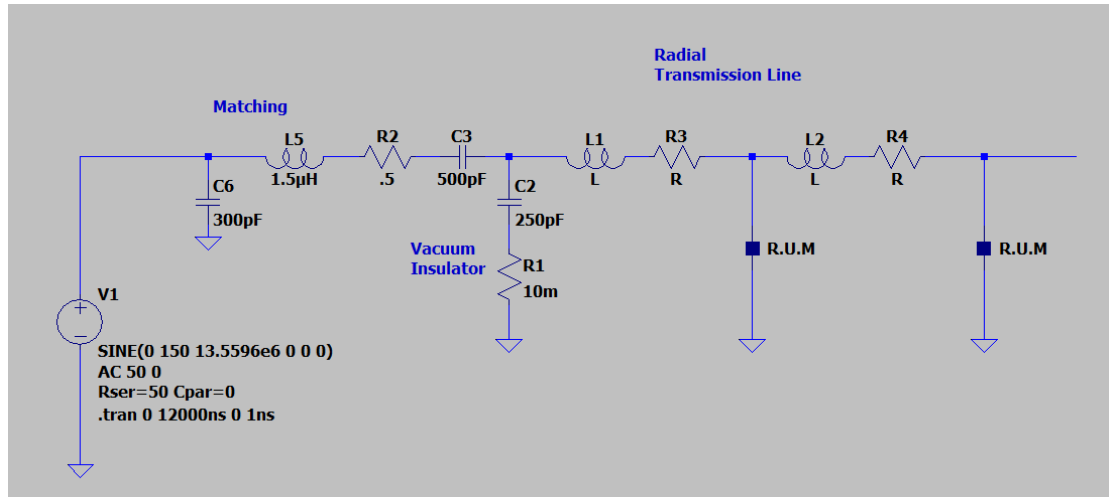


Figure 112 Riley Unified Model Schematic (Paul A Miller, 2006)

The schematic in Figure 112 of the RF drive, the match unit and the R.U.M. units models the plasma as a combination of Lieberman's ML1 model and the MEO model. *"The model is a generalization of the MEO model, with a controlled amount of inertial response of the ions added into the theory by means of an approximate first integral of the Poisson equation and a time damping procedure for simulation of the average ion response"* (Riley, 1995). These models are generalized models for the calculation of the operation of a generalized capacitively coupled plasma.

By directly measuring the signal at the input and sampling at a particular point adjacent to the plasma, a transfer function is developed in real-time for the operation of the plasma at a particular point. Using the Laplace Transform function, it is possible to derive an equation representing the transfer function of

the system. This transfer function may be examined for the combination of sinusoids and exponentials that are developed in the system and also to generate a time-domain transfer function that describes the real-time operation of the plasma for a particular operating point. This permits the precise calculation of the operating model of the system.

7.4. Oxford Instruments PlasmaLab 100 CCP System Modelling

The FRES recordings were made on the Oxford Instruments PlasmaLab 100 CCP system using the SignalHound SA44B, the balun-cabling system and the custom plain coil B-field antenna located adjacent to the chamber wall. (See Chapter 6, Figure 62 and Figure 64).

A single experimental regime was used. This involved modifying the pressure and keeping the gas-flow constant. The experiment may be summarized as follows:

Experiment	O_2 Flow rate
Pressure	30 sccm
20 millitorr	
30 millitorr	
40 millitorr	
50 millitorr	

Table 8 Experimental Regime Cross Table for Laplace Transform Generation

The FRES harmonics were recorded for each of the chamber pressure changes. The data was tagged with the recipe parameters. Using the experimental data recording the harmonics of the plasma with only a variation in the pressure, **four**

models (for each of the chamber pressures 20, 30, 40 and 50 millitorr) were extracted using the Laplace transform as described in Section 7.2 and using the values of the known elements as in Figure 111. With these models, the s -domain was swept to generate a 3D mapping of the transfer function of the operating plasma. The imaginary axis of s denotes the sinusoids of the model and the real axis denotes the exponentials in the solution. We can visualize the solution to the transfer function which is then described in the s -domain. It is possible to use this for the hyperspace for the KNN model developed earlier. The Fourier transform is merely a 'slice' of the Laplace transform. The code for the modelling is included in the appendix (see Appendix S).

The transfer function was generated for each of the parametric changes of the plasma and the inverse Laplace transform was calculated. This generates the time domain plot of the transfer function. This must be convolved with the known 13.56 MHz sinewave input to generate the output signal. The time domain transfer function is shown plotted below for the varying pressure regimes 20, 30, 40 and 50 millitorr with 30 sccms of O_2 in an Argon plasma with 100 watts of RF power applied.

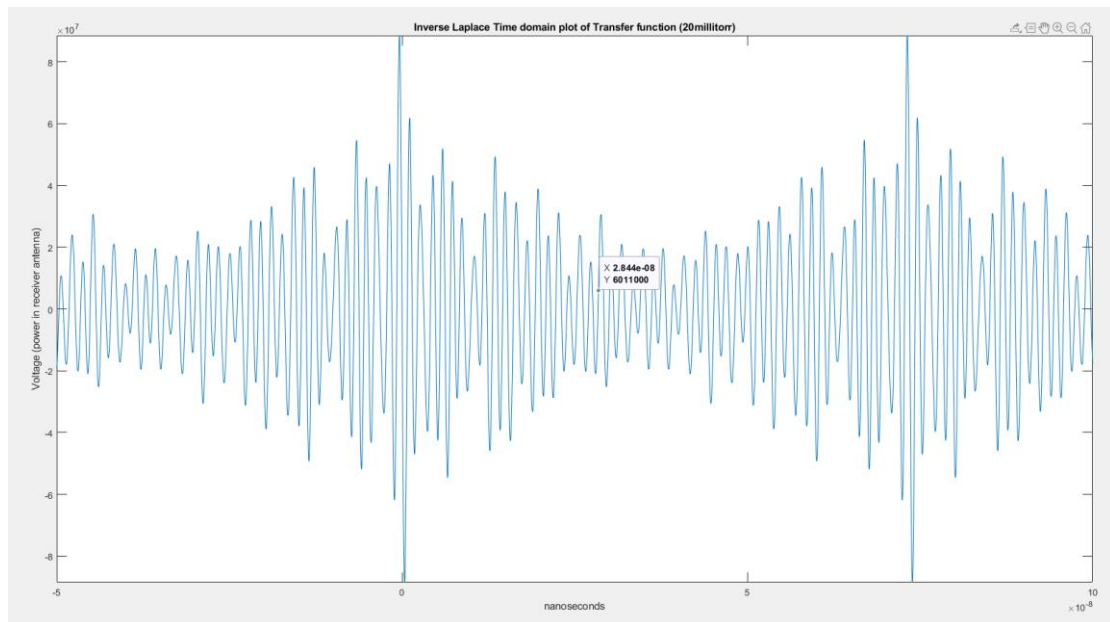


Figure 113 Time Domain Transfer Function (Chamber Pressure = 20 millitorr)

The following plots show the time domain transfer function for 30, 40 and 50 millitorr of the plasma operation:

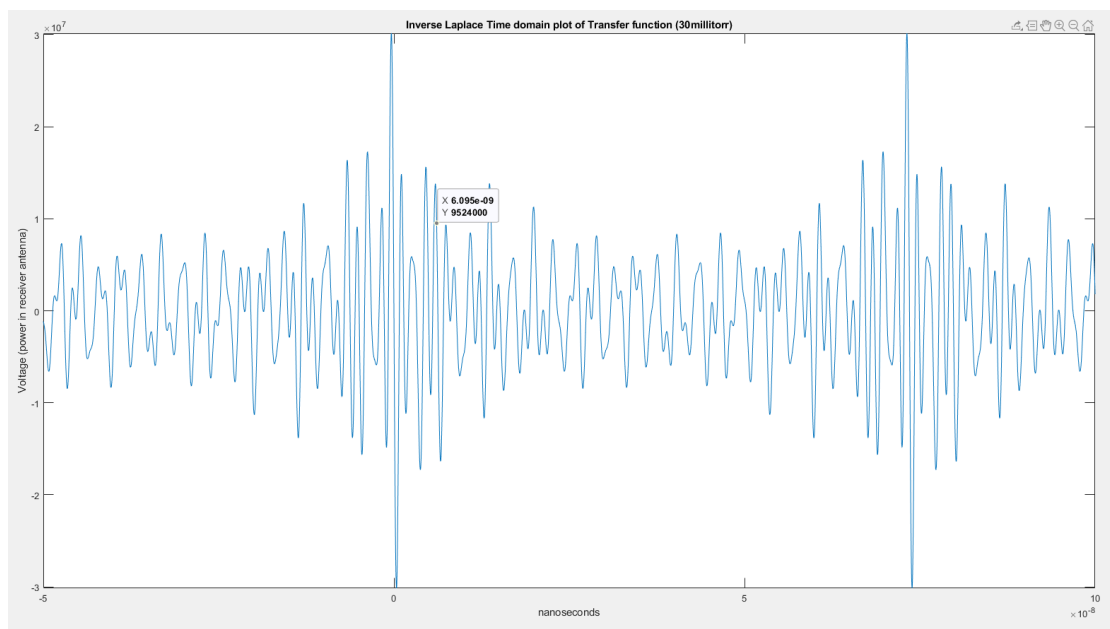


Figure 114 Time Domain Transfer Function (Chamber Pressure = 30 millitorr)

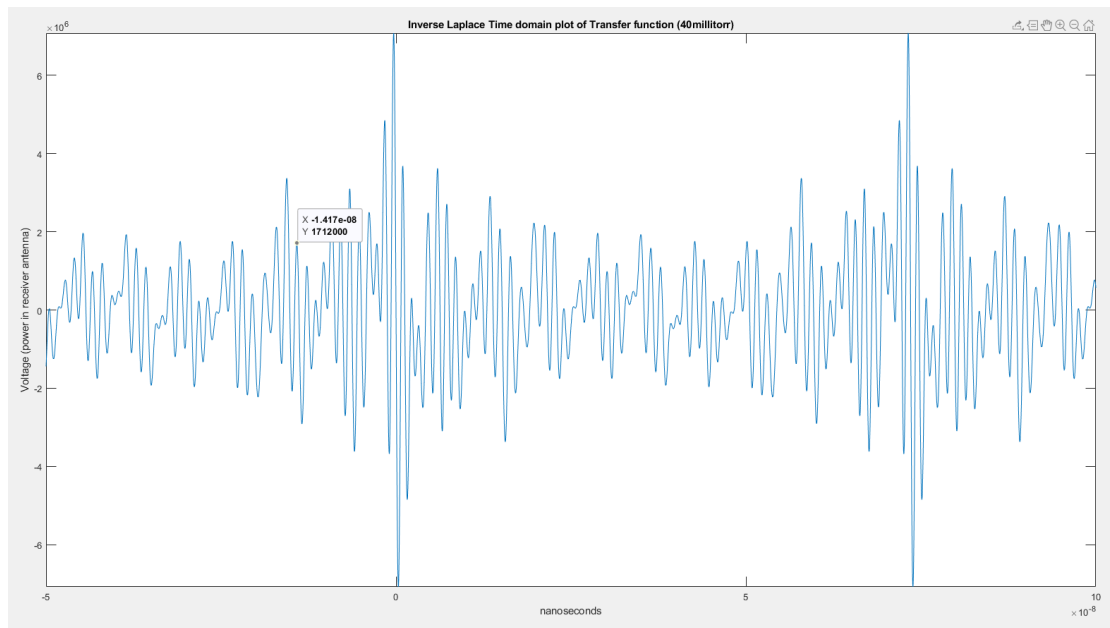


Figure 115 Time Domain Transfer Function (Chamber Pressure = 40 millitorr)

Finally, the plot of the time domain transfer function for 50 millitorr plasma operation:

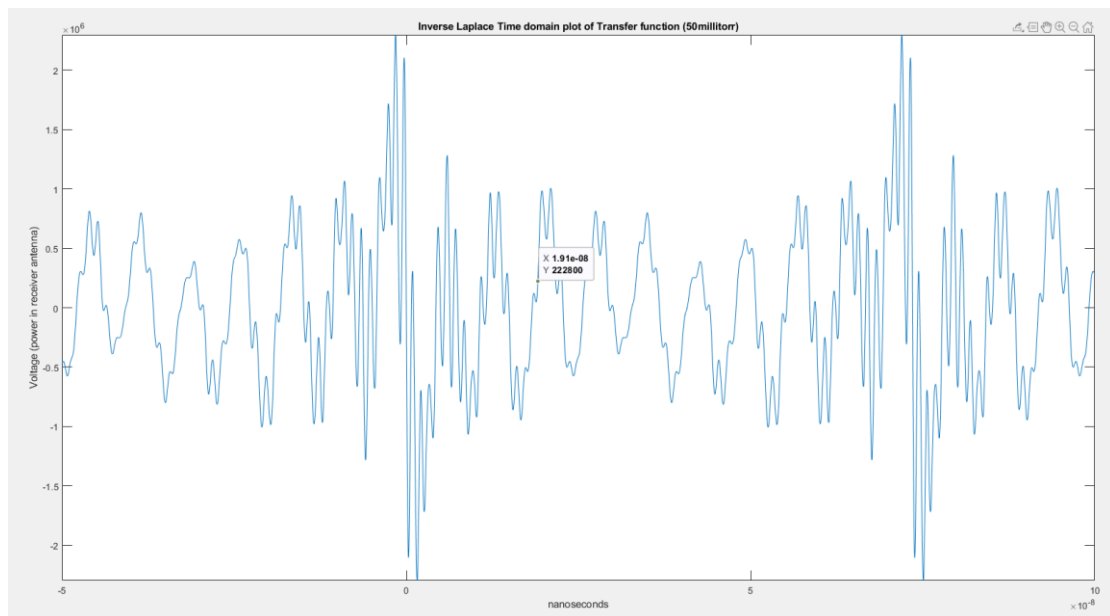


Figure 116 Time Domain Transfer Function (Chamber Pressure = 50 millitorr)

A noticeable artifact is the reduction of higher frequencies in general as the pressure within the plasma increases. In the work of Saikia *et. al.* (Saikia, Bhuyan, Escalona, Rawat, & Wyndham, 2018), the effects of the change in RF power applied to the plasma and pressure in the chamber was measured. The change in pressure from 100 to 400 millitorr increased the electron density from $1.39 \times 10^{16} \text{ m}^{-3}$ to $1.93 \times 10^{16} \text{ m}^{-3}$. There is a self-excitation of the plasma series resonance (PSR) which is triggered the moment the sheath collapses. These PSR oscillations are damped due to the strong collision damping that occurs due to the density. This in turn leads to a reduction in the amplitude of the high-order harmonics produced. This can be seen more readily when the zeroes and poles solutions to the s-domain transfer function are solved.

7.5. S-Domain Transfer Solutions

With the Laplace transform of the Transfer function, the general form is

$$Y(s) = a \cdot \frac{s}{s^2 + \omega_1^2} + b \cdot \frac{s}{s^2 + \omega_2^2} + c \cdot \frac{s}{s^2 + \omega_3^2} + d \cdot \frac{s}{s^2 + \omega_4^2} \dots\dots \text{(for 102 recorded samples)}.$$

where a, b ... represent the linear amplitude values of the recorded harmonics and ω_n represents the frequency of that n^{th} harmonic.

We can solve this for the poles and zeroes, but this will be of the form of the Fourier transform. By sweeping the s-domain ($s = \alpha + i\omega$), it is possible to determine the exponentials and the sinusoids in the solution. This provides a much more detailed model of the transfer function for each operating mode of the plasma as the

parametric changes are made (pressure in this case). Note that for an ‘applied’ method where this is being used in a real-time model of the operating plasma, the numerical calculations include very large numbers (in this case, the frequencies are 13596600 Hz and the calculations involve up to the $\sim 100^{\text{th}}$ harmonic and these values are squared and the values must be carried all the way through all the calculations). For real coefficients of the transfer function, the poles must appear in conjugate pairs and we see that in the final solution. As they are on the imaginary axis, the poles describe sinusoids and the radius describes their frequency. The real component describes the exponential amplitude of that sinusoid. Below are the plots of the solutions to the transfer functions for the changing parameter regime of pressure (20, 30, 40 and 50 millitorr). The roots have been plotted in log scale as a compression measure as the magnitudes are all positive.

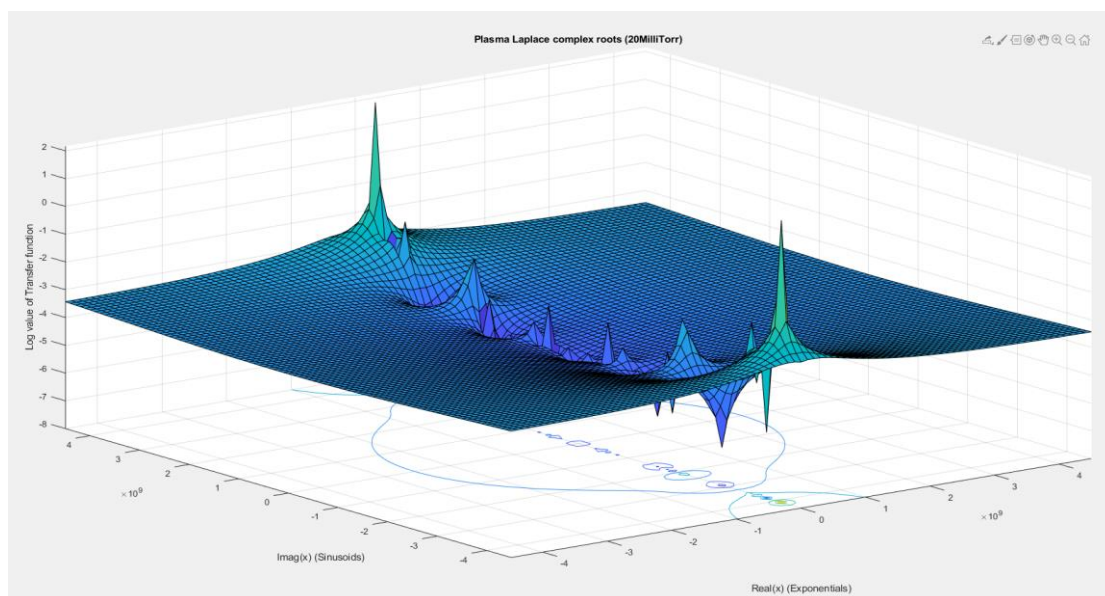


Figure 117 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 20 millitorr

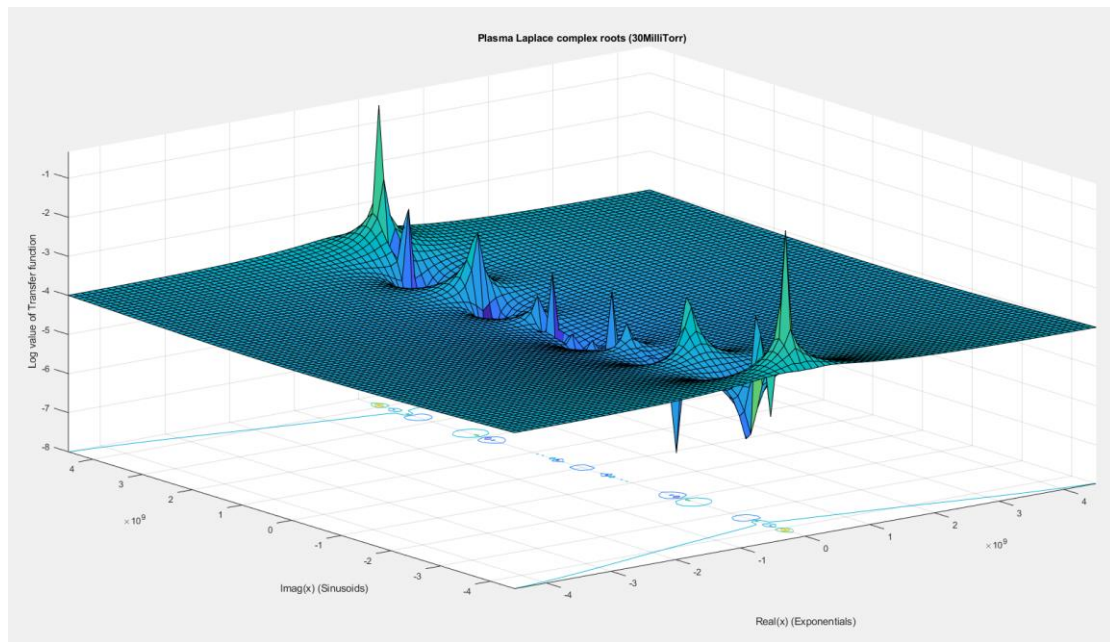


Figure 118 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 30 millitorr

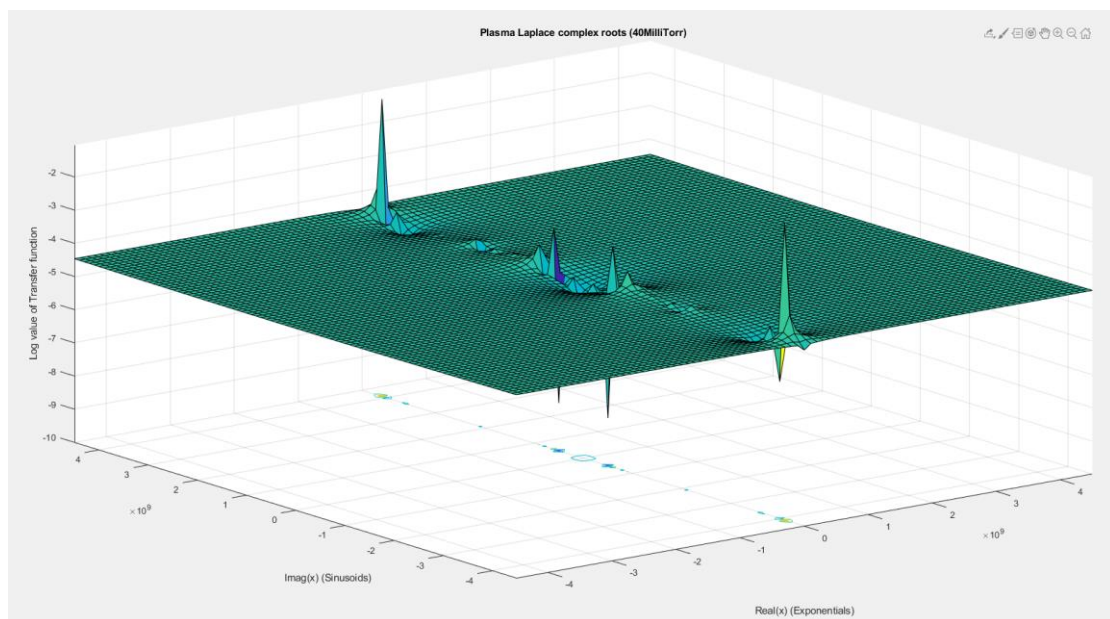


Figure 119 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 40 millitorr

Finally, the transfer function with poles and zeroes for the 50 millitorr pressure condition:

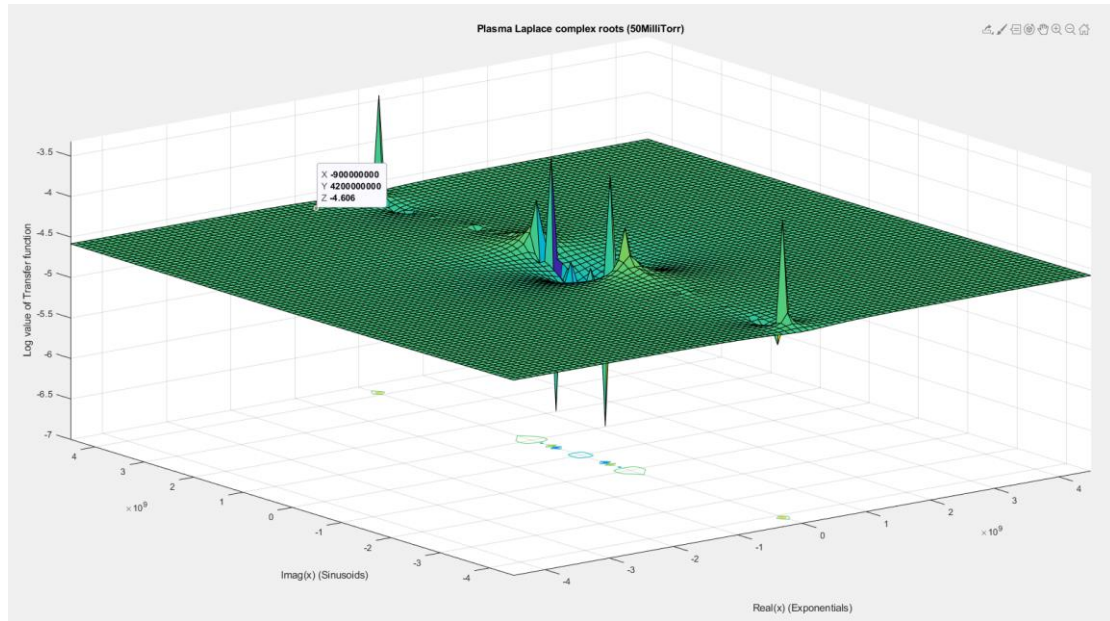


Figure 120 Complex Roots Solution for Transfer Function of Oxford Instruments PlasmaLab 100 Plasma System Operating at Pressure = 50 millitorr

For the swept s -domain, the poles and zeroes of the various transfer functions can be seen (the video is shown in the appendix) (See Appendix T) and the clear distinction between the operating modes of the plasma can be readily observed as the operating parameters are changed (in this case pressure). The symmetrical high-order (further from the centre) poles and zeroes representing the high-order frequency components are attenuated as the pressure is increased. As per Mussenbrock *et. al.* (Mussenbrock & Brinkman, 2006), “*the non-linear behavior (sic) of the boundary sheath will make the current strongly nonharmonic, with Fourier components present up to considerable order. With falling pressure their excitation will be less and less damped, and ultimately an oscillatory rf current will develop with a rms value considerably above the one expected from the sinusoidal approximation*”.

This transfer model, as an ordinary differential equation (ODE) can be utilized in a computer model to simulate the output for a parametrized input frequency. The model is not a generalized model, but reflects the operating parameters of the plasma for the applied conditions. In this way, it may be used as a step-wise indexed model to determine the operating parameters of the plasma under observation.

Further work is required to record additional information with regard to the applied frequency phase component to the input and the recorded harmonic output phase. This will permit the generation of a richer model for the transfer function which will include the phase component.

7.6. Match Unit and the Transfer Function.

The match unit of the Oxford Instruments PlasmaLab100 has a phase correcting action for the applied wave to the plasma to maximize the power transfer. The basic schematic of this unit is shown in Figure 111 above and is modelled as a phase control variable capacitor to ground followed by a series variable inductor connected to the dc blocking capacitor on the chamber active plate. The nominal values for these components were as follows:

- 300 pF for the variable capacitor to ground,
- 1.5 μ H for the variable inductor,
- 500 pF for the dc blocking capacitor to the active plate within the chamber.

Additionally, the source series resistance of 50 Ω was included. These components were included in the transfer function calculation as the transfer function was calculated from a pure-tone sinusoid of the RF drive amplifier to the measured

output at the KF40 port into the antenna and spectrum analyzer. The high order harmonics were considered shorted within the plasma as per Kelly/McNally (Kelly & McNally, 2017). We can extract the contribution by the matching unit, by modelling it as a series/parallel impedance to the main plasma impedance. In this way, it performs as a current split and series impedance between the RF source and the plasma proper. We have the impedance of the variable capacitor to ground and then the series impedance of the variable inductor and dc-blocking capacitor.

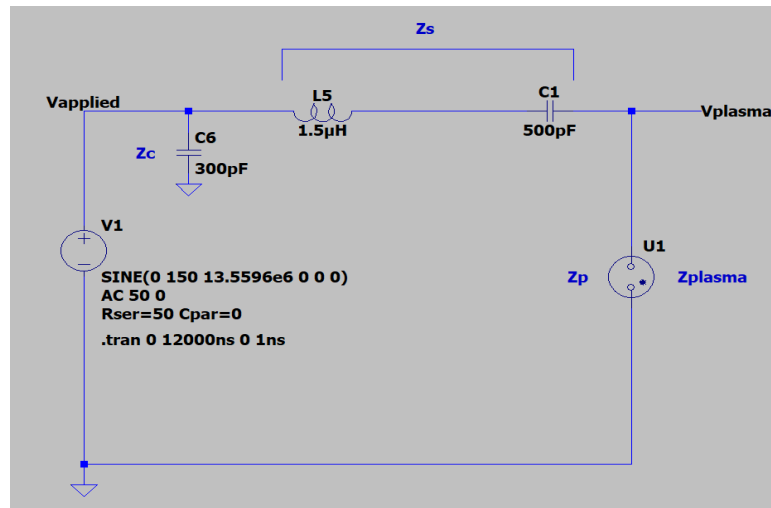


Figure 121 Impedance Matching Unit Basic Equivalent Circuit

We would like to calculate the contribution of the matching unit. The measurement is:

$$I_p = \frac{V_p}{Z_p}$$

where

Z_p is the impedance of the plasma

V_p is the applied voltage to the plasma

I_p is the current in the plasma.

The circuit for the matching unit Z_m and its approximate values are described by a parallel combination of 300 pF to GND and a series impedance of 1.5 uH and 500 pF to the plasma. Thus

$$I_{in} = \frac{\omega}{(s^2 + \omega^2)} .$$

As the match unit ensures maximum power transfer

$$V_a = Z_{total} I_{in}.$$

The Z_{total} is a simple parallel series of the matching unit and the plasma system.

This gives an equation for the transfer function including the following terms:

$$V_p = V_a \frac{Z_p}{(Z_m + Z_p)}$$

where

V_a is the source RF drive pure sinusoid

V_p is the applied voltage to the plasma

Z_p is the impedance of the plasma

Z_m is the impedance of the matching unit.

Therefore, there is a second transfer function of

$$TF_2 = \frac{V_a}{V_p}$$

If we use the default values for the variable capacitor, the series inductor and the dc-blocking capacitor and utilize a default value of 50Ω for the RF source series resistance, we can model the transfer function of the plasma.

We can then calculate the Laplace transform and using a suitable 2D mesh can extract a plot of the solution. This resulted in an additional conjugate pair of poles for the source resonance of the series LC circuit. This can be seen in the diagrams

below. The inverse Laplace transform contained a component in the s -domain so the time domain transfer response could not be readily plotted.

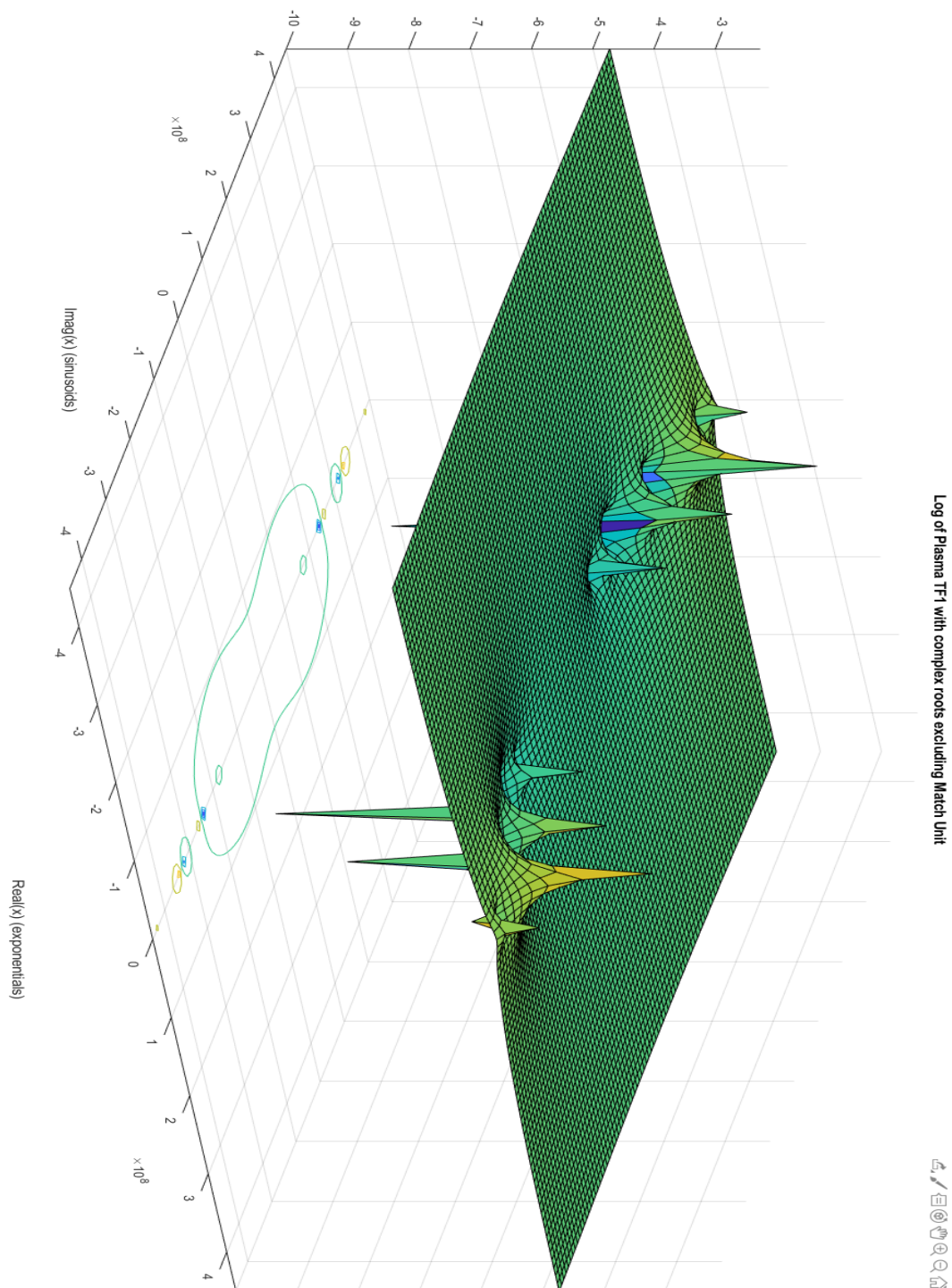


Figure 122 Oxford Instruments PlasmaLab 100 CCP Plasma Transfer Function Excluding the Calculations for Match Unit

When the calculation is made in the s-domain including the match unit, the following plot is obtained:

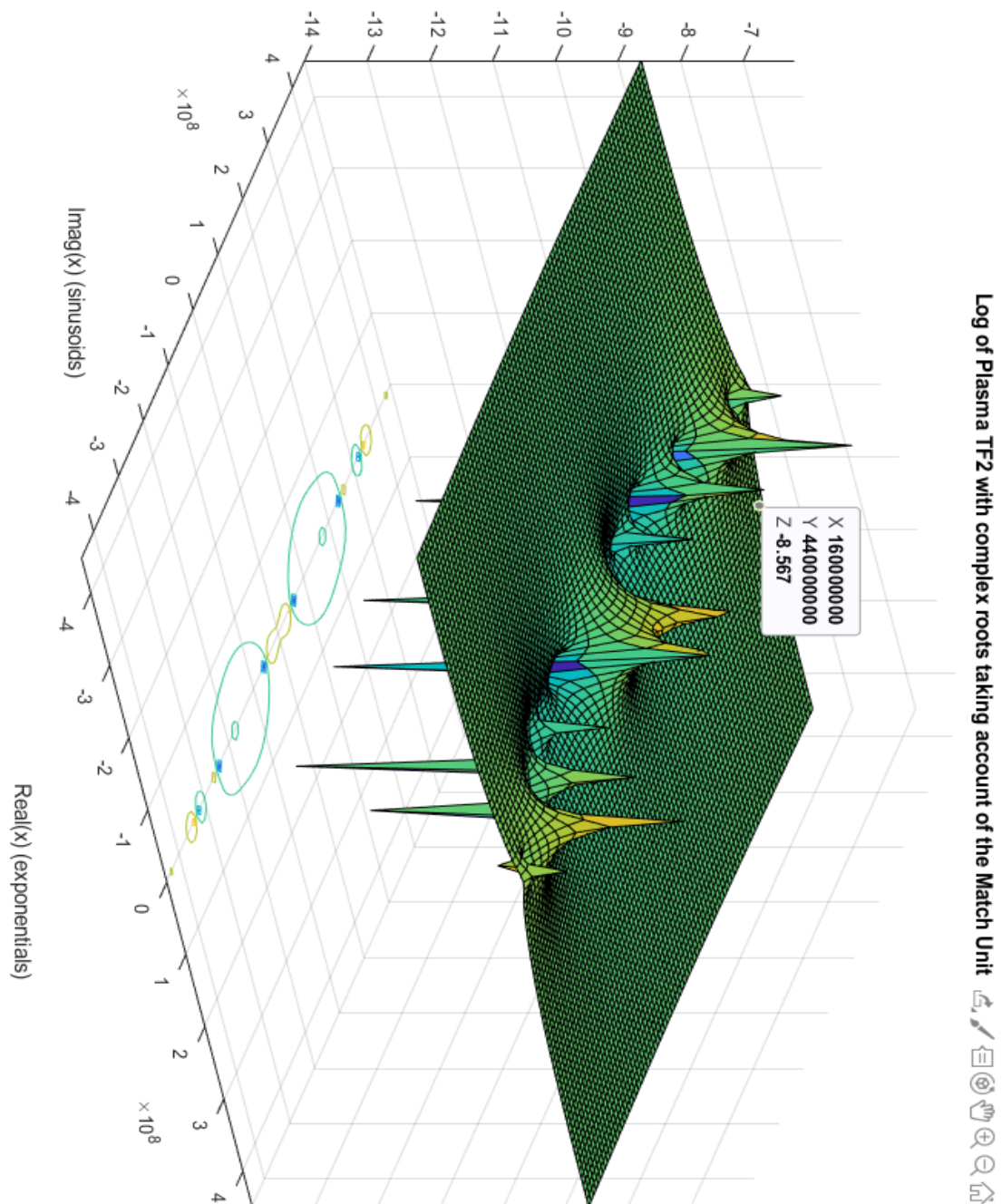


Figure 123 Oxford Instruments PlasmaLab 100 CCP Plasma Transfer Function Including the Calculations for Match Unit

The Laplace transform method provides a calculable model that may be utilized tool-side in real-time to confirm the correct operation of the plasma in an industrial environment. It is incumbent to acquire phase information in the acquisition to more fully populate these models for real-time application.

7.7. Summary

With this transform method applied, a set of equations may be developed that describes the operation of the plasma in a particular mode. These are calculable equations that may be used to confirm the correct operation of a plasma system in an industrial environment. For each of the particular recipes that would be executed on the machine, the values could be calculated to see if the machine is operating within its correct operating zone. The continuous calculation and plotting of the solutions to the equations would provide a histogram of the operation of the machine. These histograms could be used for diagnostic purposes and for tool matching in an industrial environment. These may be summarized as follows:

- Continuous plotting operating plane of machine
- Tool matching
- Correct recipe operation
- Go/No-go indication for wafer processing, event excursion

Chapter 8 Distributed Scalable Multivariable Industrial Data Acquisition

8.1. Introduction

The roots of this project lie in the monitoring of semiconductor industrial processes. The semiconductor business relies on large scale equipment deployment combined with very high-cost manufacturing systems and processes. Systems that can monitor and control these machines and increase the uptime or controllability is a very worthwhile investment. There are a number of sensors that have been built and are used at the machine level for process control. These systems are built using master-slave technologies such as SECS/GEMS communication systems. We present a novel scalable addressable network sensor system utilizing the latest Message Queueing (MQ) communication systems. MQ systems have been used in banking for redundancy and security for a considerable time to ensure the scalability and reliability of message delivery in a high-rate production environment (IBM, 2019), they have not been as quickly deployed in the semiconductor industry. The semiconductor industry has relied upon older point-to-point and master-slave serial-based communication systems to interrogate the sensors attached to machines. These ‘polled’ sensor data streams are then inserted into ‘islands-of-data’ concentrator machines. Each of these islands has their own protocols and communication methods to extract the data stored.

The design of the overall control system is to acquire the data from these sensors and contribute to a scalable monitoring system for improved process control. The major focus of this thesis is on the development of novel sensors, and the

understanding of data produced by these, rather than their implementation in closed-loop process control. “Closing the loop”, and the verification of the use of RES techniques for this, is work for future researchers. By looking across the set of machines and at each step of the process, a large amount of data is generated, which can be evaluated in real-time to generate a view of the existing process and to build a model. With the model, predictions can be made as to the correct operation of the system and determine drift trends before they reach critical levels. In the first instance, a level of analytics is performed manually in order to determine an accurate metric to locate a systematic fault. This can then be automated as analyses to detect and notify in real-time the emerging fault. With this notification, early action can be taken to reduce the impact and cost of this. With the basic cost of a 300 mm diameter silicon wafer at \$400 (Mulay, 2014) and an approximate time in production of 10 days, the earlier a wafer can be determined to be out bounds, the greater the cost saving. High-end Intel Xeon processors on a wafer can be valued as much as \$4,000 each and similarly high-end individual FPGA devices on a wafer can be valued as much as \$13,000 each. With a 300 mm diameter wafer, this results in approximately $70,650 \text{ mm}^2$ of valuable die area. The Xeon die size is approximately 413 mm^2 and allowing a perfectly circular wafer, this should yield conservatively 140 devices at \$4,000 each giving a processed wafer an approximate value of \$560,000. Any detection system that removes a faulty wafer at the earliest stage frees the slot for a new wafer start, thereby raising the effective yield.

8.2. Architecture at the Edge - Scalable Data Collection

Edge Computing, the next major paradigm of computing after cloud computing, is the use of resources at the edge of the network to augment or supplant resources in the cloud. Gartner (Gartner, 2018) estimates that by 2022, 75% of enterprise-generated data will be created and processed outside a traditional centralized data-centre or cloud. There is an obvious appeal both with the rich data from the edge and with the improved performance of the centralized resources. Resources at the edge provide an opportunity for low-latency and high bandwidth communications, lower energy consumption (albeit, there are more devices), (potentially) improved data security and the possibility to expand the analytical processing at the edge adjacent to the rich data.

Although it builds new technology components, the semiconductor industry is quite conservative in the design of the control, monitoring and analytical systems attached to the process. Between the process and the end-users of the statistics, be they process engineers, accountants or line managers, there are a variety of decentralized resources managed by individuals, support groups, tool owners, corporations and others. In an ideal world, these federated resources could be exploited in a unified manner by users and application designers. A drawback is that the current edge-components are quite heterogeneous, not just in their capabilities, but also in terms of resource ownership, security, privacy, reliability and other such soft factors (Pedro Garcia Lopez, 2015). The basic unit is the data collection sensor. Sensors are deployed across the machines to measure particular parameters. The sensors have a certain degree of accuracy, repeatability and a sampling rate. There are even sensor concentrators that already collect data on some of the plant floor machines. These comprise some of the federated

components. The data from all these sensors may be combined to build a more consistent story of the status of the machine. The Dublin City University (DCU) FRES sensor is one such sensor and there are many more on each different machine.

In what follows, we refine our federated edge-computing design and discuss how an infrastructure standardized around the Message-Queue metadata enables this ability.

- We identify a ‘platform vision’ to address the extreme heterogeneity of resources at the edge of the network.
- We build reference acquisition and processing edge-computing designs.
- We show how the layered security within the system complies with traditional conservative federated systems.
- We build a FRES sensor that has support for distributed MQ based messaging or may be directly driven by the toolside Graphical User Interface (GUI). The FRES sensor supports edge-computing and federated data collection out of the box.
- The FRES acquisition tool using the SignalHound SA44B spectrum analyzer is demonstrated using the Mosquitto™ MQ messaging system and the code is included for same in the appendix.

As well as the FRES system, the Ocean Optics USB range of optical spectrum analyzers (USB2K and USB4K (2K pixels and 4K pixels respectively)) are supported by the MQ message bus system. The code for these is also included in the appendix (see Appendix D). With the development of the bespoke FRES hardware platform using the Xilinx Zynq ARM processor, the code can be

recompiled to run on this platform and the bespoke platform will be a 'drop-in' replacement for the SA44B USB analyzer currently used. This will permit higher acquisition rates and real-time implementation of the detector technologies explained in the former chapters.

8.3. Analytics Bus for the Edge

Here we examine the platform abstraction that we developed to ease the burden of data acquisition and data analysis at the edge. The basic premise was to design sub-systems that utilize commercial-off-the-shelf (COTS) components such that the system could be installed on rich-edge node devices on existing infrastructure. As indicated, the semiconductor industry is very loathe to introduce new infrastructure on existing plant floors even if the derived data can be shown to be very beneficial. The design is a storage and communications platform that is widely distributed over heterogeneous infrastructure, whilst using accepted protocols and security mechanisms. This provides an integration of resources at the edge with decoupled data analyses, data analytics, data storage and analysis display. We examine the other requirements of the system:-

8.3.1. Homogeneous Interface

This permits application developers to acquire data from very basic to very rich resources at the edge and present this for inclusion into the overall analytic data-set. This property is important given the wide range of resources and data-types at the edge.

8.3.2. Federated Architecture.

All the edge devices (though within our security context) should be able to contribute to the resources and be part of the rich data set for analytics.

8.3.3. Locality

By placing powerful rich computing modules (in our specific case, the Intel i7 NUC with 256GB SSD and 1TB magnetic 64bit computing) at the edge and being adjacent to the acquired data, low latency, high frequency measurements may be made and processed. With the rich computing power, this enables the use of the virtual or 'container' type computing paradigm. Thus, compute loads may be orchestrated to run at the edge, adjacent to the data.

8.3.4. Administrative Boundaries

By utilizing stable proven protocols and security components, problems with the data security and tiered network access can be mitigated.

8.3.5. Secure Routing

Again, by utilizing stable proven protocols, architecture and security mechanisms, the routing of sensitive data can be controlled. Just encrypting the data is not enough. By controlling the access and using the tiered firewalls that exist on plant floors, the flow of data can be controlled.

8.3.6. Publish-Subscribe Paradigm

With secure publish-subscribe being crucial to the scalable data analysis architecture, this underpins the viability of this system. Communication and also *flexible* communication permits data flows to be readily orchestrated on existing

federated infrastructure. This is a key feature to a system that sits atop existing subsystems to acquire data, process it, store it and permit analysis of it.

8.3.7. Incremental Deployment

For a flexible, maintainable system, it must be incrementally deployable. This can be through a hybrid combination of overlay networking for deployment and then native implementation. This could be implemented via software-defined-infrastructure (SDI), but again, as seen earlier, the current (and intermediate future) network deployments on the plant floor are conservative and rely heavily on proven hardware deployments with known characteristics and ready industry support.

8.4. Existing Platforms

The current state of the art is a Manufacturing Execution System (MES) stood upon a hierarchical network with islands of automation and data-collection. For the semiconductor industry, quite a few of the MES systems are grown in-house. An example is Intel with its MES300 (Mouli, 2005), where the existing (in-house) system was re-written to build an extensible framework-based architecture. This is to permit the supporting of legacy systems and also to give a framework to build new functionality or data-analysis capabilities. *“The new system provides flexibility in modeling and executing technology development in an automated fab operations environment, thereby delivering world-class information turns crucial for rapid process development. In addition, the new system enables fully automated fab operations for efficient high-volume manufacturing.”* (Mouli, 2005)

Each tool/machine provides a rich set of data that can be processed to provide virtual metrology and, combined with data across the plant, can give a higher-order view of the overall system from the lowest level to overall plant production level. The machines comprise a heterogeneous network, for example, the Hitachi 712 ECR plasma etch systems run the Windows® XP operating system with fixed function components. As indicated earlier, the capital investment of machines for the semiconductor is very high, so there is a tendency to ‘sweat the assets’. As we have seen, only the tier-1 semiconductor manufacturers move to the very latest equipment and for some manufacturing processes, the die features do not need to be made smaller (e.g. LEDs, linear voltage regulators or legacy TTL devices), but for all the manufacturers, there is still a requirement to automate manufacturing, maximize quality-control and output.

We have seen that the Nikon Body-7 photolithography machines are still being used in the marketplace and these are upwards of 25 years old. The subsequent generation of machines based on the DEC VAX processors are upwards of 20 years old and they still command a price of \$1million when fully serviced and operational (Eptek, 2017). As for the Hitachi 712 ECR etching tool, these tools are from the early 2000s and the software is ‘owned’ by the tool provider, source code is not provided and data files describing sensor data are not explained. The MES communicates with this system through a file-exchange mechanism over an Ethernet based TCP/IP FTP (Transmission Control Protocol / Internet Protocol, File Transfer Protocol) like network. Therefore, any edge computation requirements on this node are fraught as the machine is old by today’s standards and has limited computational performance. It is not capable of performing any additional tasks and modifying the software on the machine at any stage

invalidates any support by the manufacturer of the machine. Hitachi do not have the ability to debug these production machines and merely copy whole disks from master copies to make the computer system operate. In our system, we provide the edge computing and duplicate sensors if so required. Data can still be acquired from the local node, in the manner similar to the MES, but these are used to augment the local low-latency high frequency data measurements. The scalable MQ based communication system and the modern sensors permit an older tool to be instrumented and this new multi-sensor data can be fed to the MES system. At the tool-side level, additional feedback loops and controls can be implemented to further strengthen the operation of the machine with local diagnostic displays for the operators to better manage the tools. This can be seen in reduced downtime, better tool alignment and operational status.

8.5. Internet of Things (IoT) Rich Compute Edge Platforms

We utilize the distrusted storage and compute power of the edge-computing paradigm. We build and demonstrate a Message Queueing / Subscribe-Publish network that is based on secure TCP/IP with local brokerage. The design was to form a basic message system that was flexible enough to contain the various types of data from all the sensors that could be built on relatively powerful compute nodes-on-the-edge and could traverse networks in a relatively seamless manner. By combining Time-Sensitive-Networks with Message Queueing (MQ), this permitted the scalable, dependable, data rich acquisition system to be built.

Other sensors have a similar design whereby time-stamped consecutive data readings are performed by the acquisition node and posted to a MQ system. This is the function of a MQ publisher whereby messages are published to the MQ

broker and subscribers read from the MQ broker. The MQ system forms the backbone of the transport system. The MQ publish-subscribe system can be transported across IP networks with routing configurations. These permit the scalable deployment of acquisition systems within a plant environment. As part of the project, one goal was to design the sensors to be part of larger connected system. With sensors in general now approaching their absolute limits (in both accuracy and repeatability), the methods employed are as follows: we take the output from a multitude of sensors and across a multitude of machines to analyze on a system level and thereby produce the control system which has the accuracy, predictability and repeatability required. We consider the sensors to be machines at the IOT (Internet of Things) level and design the data flow from each sensor thus. The business of semiconductor production is not just based on the output of individual machines, but the combination of all the systems, including the control and quality feedback mechanisms.

8.6. Operation of a MQ system

With MQ systems, the fundamental operation is if that a message cannot be delivered immediately, it is secured in a queue where it waits until delivery is assured. MQ messaging enables software applications and sensor subsystems to connect and scale. The sensors attached to the machine become intelligent and can connect to each other, as components of a larger application, or to user devices and data. MQ messaging while synchronous from the machine is asynchronous across the network. This decouples applications and sensors by separating sending and receiving data. It also means that with correct routing, layered or

virtual sensors may be constructed. For example, in the case of the FRES, the basic high-speed harmonic data may be published onto the MQ data bus. The database-populator may subscribe to the raw data and write it away for further processing. At the same time, a 'plasma gas-flow controller process' may subscribe to the raw data and make predictions about the gas-flow and adapt the gas-flow to close the feed-back loop. The basic flow of a MQ system is as follows

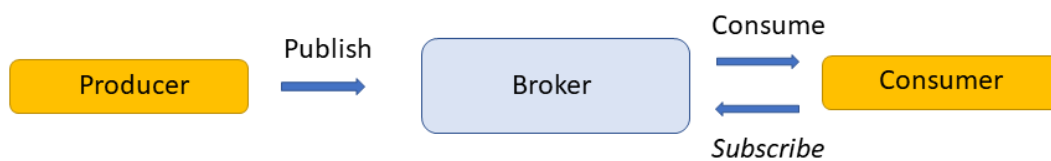


Figure 124 Basic MQ System Structure

The basic architecture of a message queue is fairly straightforward - there are client applications that can be intelligent sensors, called producers, that create messages and deliver them to the broker (the message queue). Other applications or virtual sensors, called consumers, connect to the queue and subscribe to the messages to be processed. Software may act as a producer, or consumer, or both a consumer and a producer of messages. Messages placed onto the queue are stored until the consumer retrieves them. If they are not consumed, they are stored locally to a backing-store until they are purged or they are retrieved. This basic architecture may be extended by making the brokers more intelligent. One such method is to have exchanges within the brokers. Here, messages are not published directly to a queue; instead, the producer sends messages to an exchange. An exchange is responsible for routing the messages to different queues

with the help of bindings and routing keys. A binding is a link between a queue and an exchange.

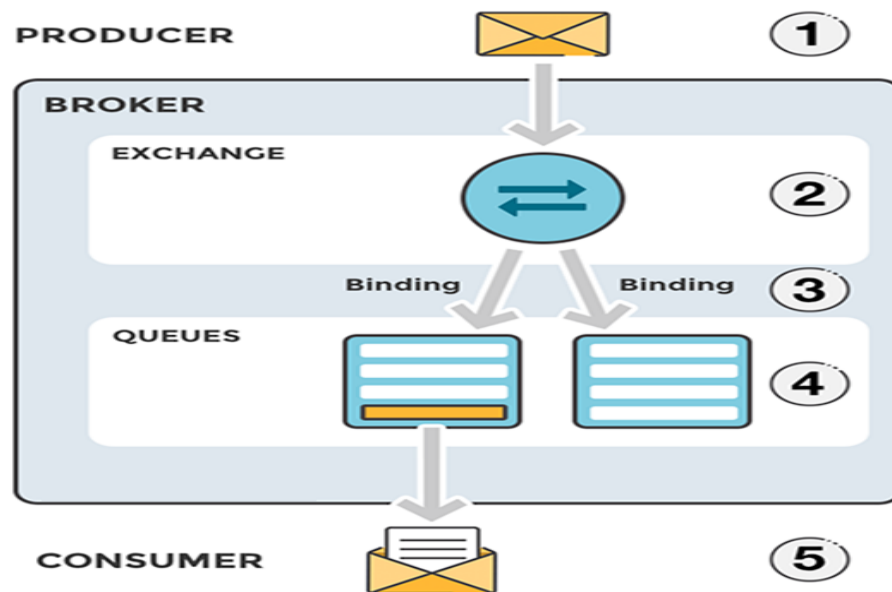


Figure 125 Example Routing Exchange Based on Rabbit MQ ®

In the case of the Rabbit ® MQ, the producer of sensor data publishes a message to an exchange (1). See Figure 125. The exchange receives the message and routes the message based on a routing key (2). The bindings must be created linking an exchange to a queue. In the example, two bindings to different queues are shown (3). The messages are routed to the correct queue based on the keys (4). The consumer, having subscribed to a queue, consumes from the correct queue (5).

The ability to connect disjoint data streams from multi-sensors tool-side on the plant floor and to be able to route them by dynamically changing the keys is a huge step forward in the design of data acquisition systems on semiconductor tools and systems. With software control, different sensors can be routed to applications that can form virtual sensors and these can be used for monitoring and closed-loop control of the semiconductor processing steps.

8.7. Network Scaling at the IOT Level

Different manufacturers have different scales for production. For example, Analog devices has a plant located in Limerick, Ireland. According to Manufacturing Operations VP in Ireland, Denis Doyle, *"We run 150 flows to make maximum use of our equipment with a run of 2000 products on our books, which are almost all made-to-order products. Volumes can vary then from 100,000 chips a year to several million"*. The plant is operated very differently from say the Intel plant at Leixlip, where there were approximately 400 Hitachi M712E etchers (now deprecated) running on the plant floor. With the FRES sensor system, the OES sensor and a Daihen Autotuner acquisition system all running on the scalable MQ bus system, the network requirements are quite staggering.

Consider the following numbers:

- Approximately 400 plasma etcher units (for a large production plant)
- At 2 Hz acquisition (and the FRES target for bespoke hardware is 100 Hz), each unit produces 4,000 data points per second (for just a single sensor)
 - FRES (100 Hz) for 120 harmonics and 106 points per harmonic
 - ⇒ 1.2M floating point numbers per second
 - TRES (5 Gsps) for V and I, before and after the match unit
 - ⇒ Stored locally, but available on request
 - OES (2 Hz) (4K unit) 8,000 points per second
- Average number of sample points stored per chamber is 7 billion
- Total number of sample points stored is of the order 3.98 trillion
- The number of database rows is 995 million

- The required I/O response time is of the order of 2 milliseconds
- 0.14% of the raw data is invalid and must be discarded, allowing for 99.86% validity.

Two very clear messages come from this, the first being that a logical approach must be taken to transport the large amounts of data from the sensors and the second is that the acquisition data does need to be retained for analysis, should a fault occur and be recognized at a later stage permitting later analysis and data mining.

The FRES system was designed to capture the harmonic data from the plasma system and then this could be analyzed at a later stage to identify the underlying principles and calculations for the plasma operation and their relationship to the harmonic content. The design was twofold, one to be a diagnostic tool for the plasma operation investigation and the other to be a deployable tool in an industrial production environment. Commercially viable prototype FRES tools are being built and sent to a major Asian semiconductor manufacturer to be evaluated onsite.

The FRES tool captures the data and may store it in a local log database; it also sends the data over a local messaging system to the broker system. This broker system logs all data to the file system. The GUI system then extracts from this broker system and displays the recorded harmonics on the screen. The FRES acquisition system can also be configured to send the data over the MQ messaging system, so that 'subscribers' may acquire the data and perform operations on it. This may be a subscriber that places the data into a database, or a subscriber that analyses the data and performs feedback control operations on the machine.

For the FRES data analysis, the data was stored locally using the inbuilt message broker system and the data was stored on the local computer. An Intel i7 NUC was used with a 500 GB hard-drive to store the acquired harmonics. With the machine on the network, the data was available over the MQ databus system also. For the investigation on the Oxford Instruments PlasmLab 100 tool, the FRES system was run in tool-side mode, where the GUI controlled the local broker and the acquisition front-end. The data was acquired and stored in raw form in a local log file, and also through the broker into structured file-based time-stamped tagged files. In 'hop' mode on the SignalHound SA44B, where we hop from harmonic to harmonic, 100 points are acquired every 250 ms. In real-time mode, this increased to 101 points every 30 ms and with finer resolution (resolution bandwidth RBW = 100 Hz), could acquire 6,744 points every 30 ms.

By using 'Computing at the Edge' and Rich Feature end-nodes, the raw data may be stored and analyzed locally while at the same time is available for later retrieval if desired. The data may also be streamed from the device if monitoring of that particular node is required. The hierarchical structure based on network infrastructure is scalable as an IOT type system permitting data from all the nodes and sensors to be available across the network.

The underlying fundamental of the system is the messaging infrastructure and the Rich-Feature end nodes. This is summarized in the figure below (Figure 126).

Data Acquisition from a Sensor Perspective

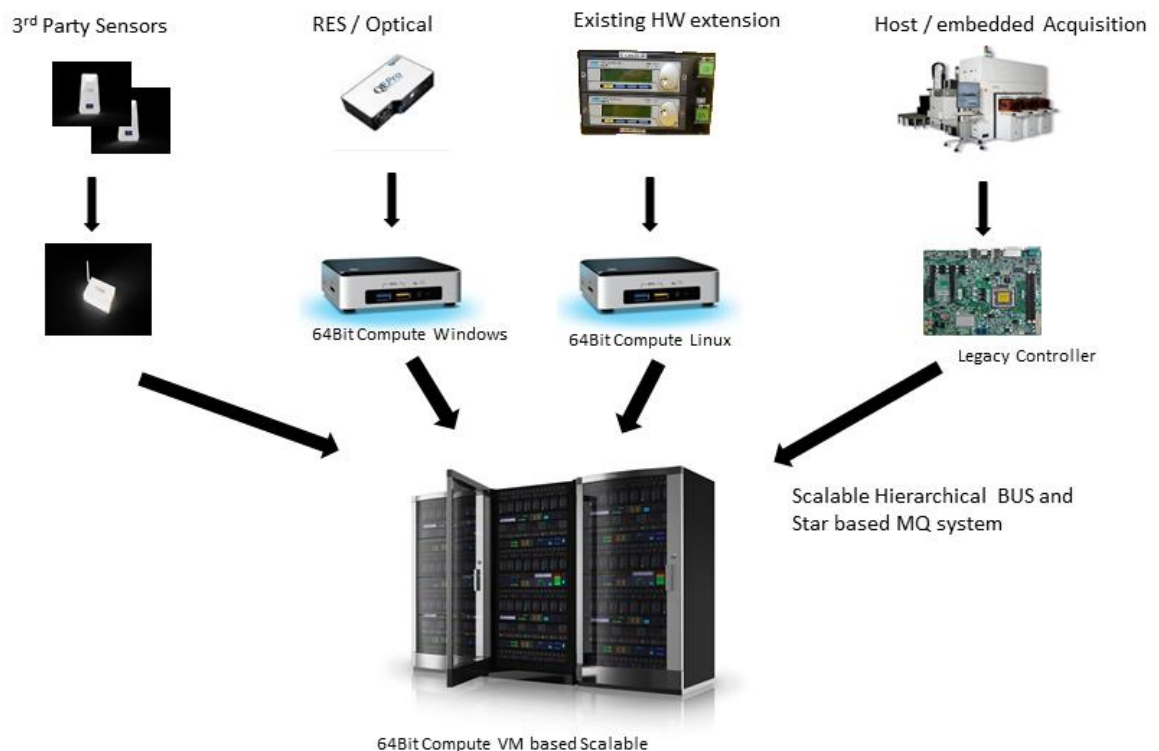


Figure 126 Hierarchical Data Flow for IOT System

The basic structure is to align the data from the sensors to a common design. In this design for the DCU FRES sensor, the basic architecture is to take a flow of data in the frequency domain in blocks of acquisition and send these on to the MQ system for insertion into the upstream database. At the same time, a feature extraction and summary generation are completed and this too is presented onto the MQ databus. In the design code shown in Appendices D, L, M and N for the scalable architecture, we used the Mosquitto Message Queuing (MQTT) library as the underlying data transfer protocol. We then designed and built our own data sensor protocol on top of this basic communication system.

“MQTT (Mosquitto) is an (Open Source) Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for the communication in Machine to Machine and Internet of Things contexts where a small code footprint is required and/or network bandwidth is at a premium.” (HiveMQ, 2018)

The protocol is a basic publish/subscribe architecture and we define a novel protocol on top of this to provide for the acquisition and control of the machines using the sensors such as FRES, TRES, OES and the like to handle the scale of the manufacturing plant as outlined above.

8.8. Sensor Protocol for Scalable Data Acquisition

As described above, the use of plasma chambers is a vital processing step for semiconductor manufacturing. As we have seen, the larger plants may have significant numbers of machines operating at the same time. With the FRES system acquiring multi-point data across a large number of harmonics at a relatively high acquisition rate generates a large amount of data per machine. To analyze across all these machines operating in parallel, a scalable MQ based architecture was designed. One of the goals of data-mining the acquired data in the database is to determine if the variability is inter- or intra-wafer. Data analytics is used to close the feedback loop of the process and reduce the variability of the system as a whole. To perform the data analytics, data acquisition is first required. With multi-sensor acquisition, this variability can be calculated across all the data sources. In this way a virtual metrology can be determined for the operation of any particular

process. The FRES data can be combined with the TRES data, the OES data and any parametric data that may be available from the SECS/GEMS data on the machine. This data may also be correlated with any other metrology data that may be available from other sources such as end-of-line test processes or procedures. The FRES, the OES and the autotuner sensor subsystems as included in Appendices (see Appendices E, D and N) utilize this messaging system to traverse the network and deliver the data to the database populating tool (Not included in this thesis as a database storage facility is a requirement for a production level system. For data analyses in the preceding sections, we used the tagged time-stamped CSV files as they were readily interfaced to the analysis tools).

8.8.1. Scalable Hierarchical Acquisition Protocol (SHAPer)

The message format herein describes a flexible compact method of transferring acquisition data onto the MQ subsystem and later retrieval by the database populating tool for analysis. The format of the messages loosely resembles the insertion data format into the columns of the database, but is in a richer format than more the straightforward TSDB (Time Series Data Base) series data. The data ranges from the various sensors are quite different and thus have quite differing data formats. The format has to be able to handle 'blobs', comma-delimited and fixed format data. There is also a requirement for more modest data description. Binary encoding permits the viewing of the data directly, and whilst some data is viewable, a high proportion is not so. There is also the requirement to minimize redundancy. In this manner, the format permits an ASCII representation of the Header, whereas the data contents may be binary and the final Tail is ASCII. Thus,

it is possible to distinguish these parts to a certain degree. (A Tail could be embedded in the binary data section).

8.8.2. Description of Base Components SHAPer

This describes the basic message format for transferring data messages over a MQ system for communication between modules within the Shaper subsystems. The basic data format is a C type unsigned char array with a pointer to the start and its length. This describes the basic message.

```
uint8_t message[];
```

```
uint32_t length;
```

The message comprises three main components:-

- The Header (variable length, may be parsed in ASCII).
- The Data section (variable (but known) length, format variable payload).
- The Tail section (contributes to integrity of data and header section).

These sections will be covered in more detail below.

8.8.3. Sections of SHAPer

The following outlines the three components making up a 'message' on a Shaper MQ queue:

The basic message is

“[Header][Data][Tail]”

Note that the '|' character is included literally in the message buffer. This character (0x7C) acts as a clear check identifier between the sections. There should always be three of them in a message. Any message parsing where there is a failure to decode these means that the message is to be discarded.

8.8.3.1. Header

The header is designed to be processed as ASCII to permit some visualization of the data.

The format is 'semicolon (;)' (0x3B) delimited strings with fixed number of records. Some of the records are enumerated (as ASCII strings). The header comprises the following components. (Note that there is No NULL terminator delimiting a string).

Char sensor_name[]

Char time[] "YYYY-MM-DDTHH:mm:ss.ssssSxx:yy"

Date time is in the format as per

<http://www.w3.org/TR/NOTE-datetime>

YYYY-MM-DDThh:mm:ss.sTZD (e.g. 1997-07-16T19:20:30.450+01:00)

Note the 'T' that appears literally in the date-time string.

YYYY = four-digit year

MM = two-digit month (01=January, etc.)

DD = two-digit day of month (01 through 31)

hh = two digits of hour (00 through 23) (am/pm NOT allowed)

mm = two digits of minute (00 through 59)
 ss = two digits of second (00 through 59)
 s = one or more digits representing a decimal fraction
 of a second (nominally 3 digits ie milliseconds)

TZD = time zone designator (Z or +hh:mm or -hh:mm), S representing
 sign (+ or -), followed by two digit hours, ‘.’ and then two digit minutes.

Char type[] “DESC,DATA,INFO,STRT,STOP,BLOB, CSV .. etc” , NOT
 empty

Char structure[] “char,uchar,int8_t,uint8_t,
 short,ushort,int16_t,uint16_t,
 int,int32_t,uint32_t,long,ulong,
 int64_t,uint64_t,
 float,
 float8,double,
 float16,longdouble
 cfloat,
 cdouble,
 longcdouble”,
 possibly empty (depends upon type[] structure above)

Char elements[] “ASCII integer representation of the number of data elements
 (max 32K, min 0)”

Char separator [] “, \ |” may be blank. If the number of elements is 0 (ie a ‘STRT’
 command) or if there is a single “BLOB” then it may be empty. This indicates the

delimiters in ASCII presented data and can consist of multiple characters that provide unique separation structures e.g. “,\”. Note re “”.

Char length[] “ASCII integer representation of the length (in bytes) of the data section (max 32K, min 0)”

Thus, when you combine these components, the header is made up of variable field lengths in ASCII with a ‘;’ delimiting the fields. Thus, each of data in the fields defined above (e.g. sensor names or separators and the like may **not** contain a ‘;’. The following shows some examples of the format:

Example 1

This is a single binary blob with no visible inserted delimiters. It is simply an array of 4 byte integers (or floats etc.).

```
UInt32_t __attribute__((packed,aligned(1))) data[27] ;
```

Header =

```
“mysensor_name;2016-10-16T12:34:44.567Z;BLOB;uint32_t;27;108;”
```

This breaks down as follows:

Sensor name is “mysensor_name” (followed by ‘;’)

Date is 16-Oct-2016 12:34:44.567 UTC (followed by ‘;’)

Data type is “BLOB” (followed by ‘;’) (108 bytes = 27 units of INT32)

There are 27 elements (no delimiter specified, binary data) (followed by ‘;’)

The length is 108 bytes (in this example, 4 bytes per element) (the corresponding receiver must have some knowledge of the intrinsic data format within the messages) (followed by ‘;’ and then ‘.’ to indicate end of header)

Example 2

This is a single binary blob with 27 off 4byte(int32 or float) records each separated by an ASCII “,”. Note that there are only 26 “,” between the 27 records, hence the length of 134.

Header =

“mysensor_name;2016-10-16T12:34:44.567Z;BLOB;float;27;134;”

This breaks down as follows:

Sensor name is “mysensor_name” (followed by ‘;’)

Date is 16-Oct-2016 12:34:44.567 UTC (followed by ‘;’)

Data type is “BLOB” (followed by ‘;’)

There are 27 elements (delimiter of ‘,’ specified, binary data) (followed by ‘;’)

Each of the 27 elements is delimited by one of 26 commas “,” (The final “:” marks the record end)

The length is 134 bytes (in this example, 4 bytes per element (float single precision)) (the corresponding receiver must have some knowledge of the intrinsic data format within the messages) (followed by ‘:’ and then ‘:’ to indicate end of header). That is:-

27 X 4 bytes + 26 X “,”

“[byte3][byte2][byte1][byte0], [byte3][byte2][byte1][byte0]

, [byte3][byte2][byte1][byte0]”:

The single “: colon at the end indicates the Data record end. In this case the data is Little Endian binary float single precision.

Example 3

This is an array of comma separated values where the delimiter is multi-character.

The record comprises 3 records of float sent as ASCII with the delimiter “,\”

Header = “ice121/at0;2016-10-16T15:36:37.0-05:00;CSV;float;3;,\;21;:”

Data = “13.234,\0.123,\4.25”: (the inverted commas are Not transmitted)

This breaks down as follows:

Sensor name is “ice121/at0”

Date is 16-Oct-2016 15:36:37.000 Eastern Standard Time

Data type is “CSV”

There are 3 elements (ASCII float) (comma delimited values)

Each element is separated by “,\”

The length is 21 bytes (in this example ASCII variable bytes per element) and the corresponding receiver must have some knowledge of the intrinsic data format within the messages.

8.8.3.2. Data

The Data section is free format data, and may be sent as binary (using designations in the header field, see examples), delimited ASCII or binary values, or fixed format fields. The length field in the header above determines the bytes within the whole message that make up the data section. The data section is terminated by a single ‘:’ character. Note that for binary data, this may exist within the data, so a linear

search may return false 'finds'. The length field is used to correctly determine the end of the data and see if a ':' follows immediately

```
UInt8_t data[]
```

```
Char ':'
```

8.8.3.3. Tail

The tail section is a simple terminator to check for correct termination of a message. It comprises a simple mod256 checksum of the header and the data (including the final data ':'). This number is stored as the ASCII (positive) representation of a byte ("0" to "255"). The number is right justified to 3 digits (e.g. '009').

Sum = all bytes of (header + data) (including ':' between Header and Data and ':' between Data and Tail.)

The calculated checksum value and the message value should match. If there is a mismatch, the message should be discarded.

8.9. Hierarchical / Tree Database

For the FRES acquisition system, there is relatively little structure of the raw data from the device. As with the OES and, for example, an autotuner, the data represents time-series-data with some tags to indicate the sensor parameters. The main concern is the analysis across the data and also vertically. There is not a requirement to track the individual data once the analysis decision has been completed. The hierarchical database (DB) concept originates from the 1960s,

with IBM and early mainframe database management systems (DBMS). Here the data is stored in a tree-like structure with records connected through links. For high-volume data, the key analysis is not in the multi-varied records, but in the exceptions of the data within the records. The hierarchy forms a tree like structure and this is a special case of the Directed Acyclic Graph (DAG). In the particular case, the Table might be:

```
Create Table machine (  
    Name varchar2(100),  
    Sensor1 varchar2(100),  
    Sensor2 varchar2(100),  
    Sensor3 varchar2(100)  
);
```

Should a machine have a sensor that is common to a number of machines e.g. local barometric pressure, then the machines form an Adjacency List. In particular we are not concerned with the closed relation of the sensors as it is possible to perform analysis over the data and then determine the adjacency. Should analysis show some form of correlation between the barometric sensing over the machines and the primary fault analysis, then it is a trivial matter to decouple the adjacency. In the simplest sense, the data is stored in a relatively 'flat' manner, i.e. all data for all machines with clear timestamps and clear identifiers. The relationships are not a fundamental part of the data analysis as they are inherent in the data-acquisition whether in time or locale.

The fundamental form is a time series database (TSDB). In the initial implementation, the data was stored as records of 'blobs' in a relational database. The only relations selected were the temporary records formed on a summary across 'machines' (sensors) and upwards in time. Later iterations moved to more straightforward TSDB systems (influx) for performance. TSDBs are used with keys based on time and streams of data. The database is optimized for time-stamped or time-series data. For example, with the Optical Emission Spectrometer, the data output is 2,000 samples every 0.5 second with 32 bit floating point values. With the FRES system, in real-time mode acquiring a single harmonic with RBW set to 100 Hz, the sensor acquires 6,744 samples every 30 ms. This is sent as readable ASCII over the collection bus, comma delimited. As we have seen earlier, this amounts to considerable data to be stored.

Plant:Location:Machine:Sensor:Time:Description,"block of values"

For all the Plants of (perhaps particular recipe), for all the Locations (perhaps within a region), for all the Sensors (of a type) : from time A to time B; analyze with respect to variation in wafer metrology.

The main function of the TSDB is to form analytics of the acquired data over time. By asking questions over a known time, variations can be determined. This can permit the viewing of trends and deviations. By viewing the data over a significant period, and across a number of sources, data that has significant noise in its signal to noise ratio (SNR) can be analyzed.

8.10. Summary

We have shown how multi-sensor data can be combined to derive virtual metrologies, how the tools and processes can be augmented by localized high-performance at-the-edge non-invasive sensors and how this vast amount of data can be hierarchically stored and analyzed. We have built the novel FRES sensor using this design methodology and this has permitted the prototype unit to be industry-ready from the test laboratory to the production floor. The prototype units have been assembled and are being deployed in an industrial semiconductor plant. This is a significant validation of the FRES technology and its implementation.

Chapter 9 Future Work Beyond PhD Studies

9.1. Introduction

The research undertaken here has led to a number of paths each with its own bifurcation. There have been decisions at many stages, where a number of factors have contributed to a particular path being taken. Not all are correct, but given the factors available at the time, led to the various outcomes. A number of areas of additional research are already presented throughout the text but it was decided to narrow the focus to build solutions that could be directly applied to industry.

9.2. Development of Non-Invasive Sensors

The TRES has direct application to the arcing events in DC plasma deposition systems such as the Endura DC plasma system. This is a widely deployed system with approximately 4,500 units in service. A real industrial challenge is that when an arcing event occurs there is no notification and the wafer is destroyed. This is not discovered until parametric testing is performed at the end of the process run. This detection may not occur until well into the production cycle of the particular wafer. The damaged wafer is carried all the way through the process which wastes 'production slots' and time. This has the effect of reducing plant yield. The development of non-invasive sensors, such as visual, spectrographic (radio, optical), exhaust gas analysis, machine-data extraction (through software means), environmental factors, etc. would permit the retro-fitting to existing plant. The sensors themselves may not be very accurate, but when combined together may produce a very rich picture of the state or at the very least may show exception anomalies. In the latter case, early detection may be sufficient for significant cost savings. An initial feasibility investigation was carried out on the operation of the

Hitachi 712 ECR plasma chamber and the non-invasive sensors that could be attached. A brief discussion of some of these is outlined below.

9.3. Pseudo 3D Planar Optical Observation.

The Hitachi 712 model utilizes a vertically mounted ECR chamber with a 2.4 GHz excited plasma within a contained magnetic field. At the top of the microwave introduction horn, there is an observation port. An optical mount was fashioned for this port to mount a C-mount lens camera. The design is to use an optical filter to narrow the observed spectrum. In this way, the plasma could be observed over the whole wafer at a 30 Hz frame rate.

9.4. Spectral 2D Observation.

By changing the filters, different observations could be made to known gas spectral groupings. This would give an observation of the species generation over a 2D view. By combining with mechanical or electro-mechanical filter changing, the different spectra species may be observed showing the 2D placing of the plasma species.

9.5. Phase-Locked Pseudo 3D Observation.

The wafer chuck has an AC and DC bias. The DC is used to both hold the wafer and to bias the plasma for correct operation. The AC is used to move the species to and from the surface of the wafer and effect the required chemical reaction. The nominal DC bias is of the order of ~ 250 V and the AC bias is superimposed. This AC bias is at an operating frequency of 2.4 MHz. The camera acquisition is connected to a phase-locked delay line being sourced by the bias amplifier for the

chuck at this 2.4 MHz rate. In operation, the bias moves the electrons at a very high rate, but the heavier ions are moved at a lower rate towards and away from the chuck based on its electric field. By triggering the camera at different points along from the zero-crossover of the AC bias, time varying observations could be made of the plasma and the location of the species.

With specific reference to the core work in this thesis, the further potential for the deployment of RES techniques needs to be investigated. The list below is a brief summary of a number of topics which could form the basis for this further work:

- Zero-IF CCP acquisition for transient detection.

For the Oxford Instruments PlasmaLab 100 CCP, with its RF drive frequency of 13.56 MHz, we have seen the efficient detection of transient events with the production of frequencies up to 500 MHz. A zero-IF acquisition system for this range of frequencies would provide a high-speed, minimum-delay real-time transient detection and analysis system.

- FPGA DSP integration for Hilbert/Wavelet for transient detection.

With the zero-IF acquisition subsystem, the output of the analogue acquisitions could be streamed to a FPGA subsystem with is programmed with the Transforms and AI classification subsystems as outlined in Chapters 6 and 7.

- Integration of ancillary measurements in PCA, magnetic, vibration, humidity, barometric for a production environment.

We have seen that the combination of multi-variate data is beneficial in characterizing the operation of systems. By collecting primary and

ancillary data, the precise operating point and process deviation is may be more readily determined.

- Phase resolved FRES with the powered electrode of the Oxford PlasmaLab 100 CCP.

The primary RF drive is the key source and using it as a time-base for subsequent measurements is of prime importance. We have seen the 'hot and cold' electron beams contributing to the sheath-current and also the spatio-temporal components of the wave propagation in the sheath edges. With the primary drive frequency, these may be resolved in a temporal and spatial manner.

- Phase resolved FRES for shielded E-field and B-field.

The basic operation would be to acquire the B- and the E-field in a non-invasive form and generate a two-component vector description of the E- and B- field and also the phase relationship between both as the RF cycle is applied to the plasma. This would enhance the transform methods that are described and would permit a greater hyperspace description for the AI classification methods for the operation of the plasma.

- Multi-window FRES and TRES acquisition.

With a multi-window acquisition, the relative spatial component of the RES system could be examined for varying positions of pairs of antenna systems. This would provide a differential measurement framework which would provide a richer data-set than the absolute values acquired by a single acquisition subsystem.

- Zero-crossover variable delay acquisition of B-field and E-field for determination of timed emission from sheath.

By using a zero-crossover detection system on the primary RF drive, it would be possible to acquire a timed relative acquisition of the RES and develop a short-time FFT to describe those frequencies that are developed at the different junctures of the RF cycle.

- Build SPICE® models of Oxford PlasmaLab 100 CCP for operation with main constituent gasses and iterate for different gasses / operating parameters. Compare with empirical data.
- Investigate the Hall effect for B-field antenna such as to make a large planar active antenna at view port, i.e. direct current measurement with an “active device.”

This page intentionally blank

References

- (CGPM), G. C. (2019). Information for users about the redefinition of the SI. *Bureau International des Poids et Mesures* (pp. 1-2). Bureau International des Poids et Mesures.
- 393, N. T. (1970). *A method for designing multi-screw waveguide tuners*. US: US Dept of Commerce National Bureau of Standards.
- A. Metze, D. W. (1986). Applications of the physics of plasma sheaths to the modelling of rf plasma reactors. *Applied Physics*, 60, , 3081-3087.
- Ahmed Shamsul Arefin, C. R. (May 15 2014). *GPU-FSk NN: A Software Tool for Fast and Scalable KNN Computation using GPUs*. Newcastle: Newcastle.edu.au.
- Al-Shrouf, L., Saadawia, M.-S., & Soffker, D. (2012). Event Detection using multisensor fusion and filtering techniques based on CWT and SVM. *6th European Workshop on Structural Health Monitoring Fr.2.D* (pp. 1-8). Dresden: University of Duisberg-Essen.
- Andrew Tan. (2016). Electrocardiograms: QRS Detection Using Wavelet Analysis. *Medium.com*, 1.
- Anton Shilov, I. C. (2018, August 27). *GlobalFoundaries stops all 7nm Development Opts to focus on Specialized Processes*. Anandtech Semiconductors_CPUS_AMD_GlobalFoundaries_7nm_7LP.
- Ariane Ferreira, A. R. (2009 June). Virtual Metrology Models for predicting physical measurement in semiconductor manufacturing. *Advanced Semiconductor Manufacturing Conference, 2009. ASMC '09. IEEE/SEMI* (p. 7). Gardanne France: Ecole Nationale Supérieure des Mines de Saint-Étienne.
- Beehive Electronics. (2018). *100 Series EMC probes*. Sebastopol, CA: Beehive Electronics (Rev 2.2).
- Beehive Electronics Inc. (2019). *101A EMC Probe Set specifications*. Sebastopol, CA: Beehive Electronics Inc.
- Bohm, D., & Gross, E. (1949). Theory of Plasma Oscillations. A. Origin of Medium-Like Behavior. *Phys. Rev.* 75, 1851, 1851-.
- Bolandraftar, S. B. (2013 Vol3 Issue 5). Application of K-Nearest Neighbor (KNN) Approach for PPredicting Economic Events. *Journal of Engineering Research*, 605-610.
- C. Woelfel, M. O. (2019 (Oct 8)). The Multipole Resonance Probe-based controller: a technology to investigate plasma-based deposition. *J Inst (Published by IOP Publishing for Sissa Medialab)*, 1-14.
- Cao, M. (2016). Electron heating enhancement due to plasma series resonance in a capacitively coupled RF discharge. *Jpn. J. Appl. Phys.* 55 096201, 1-6.
- Chapman, B. (1980). *Glow Discharge Processes*. New York: John Wiley & Sons.
- Charles Capps, Delphi Automotive Systems. (2001, August 16). Near Field or Far Field. *EDN*, 95-102.
- Chen, F. F. (2016). *Introduction to Plasma Physics and Controlled Fusion*. Heidelberg : Springer International Publishing.
- Collins, T., Getz, R., Pu, D., & Wyglinski, A. (2018). *Software-Defined Radio for Engineers*. Norwood, MA: Analog Devices, Atech House.

- Commission on Physical Sciences, M. a. (1991). *Plasma Processing of Material: Scientific Opportunities and Tecnological Challenges*. Washington DC: National Academy Press.
- Daihen Corporation. (1995). *Instruction Manual for 2.45GHz 2.0KW Automatic Microwave Tuner SMA-20A Ver 2.1 & CMC-10A Ver 1.3*. Daihen, Ref No. LJ-L5-0595A.
- Dixon-Warren, S. (2018, May 25). Revisiting-the-seminal-bosch-deep-reactive-ion-etch-patent. *Techinsights*, 1.
- Doyle, D. (2017 Mar 29). <https://www.limerick.ie/business/why-limerick/success-stories/analog-devices>. *Limerick Business*, 1.
- Dvorak, P. (2013). Modelling of electric characteristics of capacitively coupled discharges including nonlinear effects of sheaths. *Plasma Sources Science Technology*, 1-10.
- E. Gogolides, H. S. (1992). Continuum modeling of radio-frequency glow discharges. I. Theory and results for electropositive and electronegative gases. *Journal of Applied Physics*.
- Ellingson, S. (2003). *Correcting I-Q Imbalance in Direct Conversion Receivers*. Virginia: Virginia Tech ECE.
- Eptek. (2017). *Nikon Body 10 to Body 14*. Hutto, TX.
- Frederik Schmidt, J. T. (2019). A generic method for equipping arbitrary rf discharge simulation frameworks with external lumped element circuits. *Applied Physics* 173106, 125.
- Frederik Schmidt, T. M. (2018). Consistent simulation of capacitive radiofrequency discharges and external matching networks. *Plasma Sources Science and Tecnology*. 27 (2018) 105017 (7pp), 1-7.
- Gabor, D. (1944). *Theory of Communication*. Toronto: Toronto University.
- Gartner. (2018). <https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/>. Gartner.
- GE, L., & Zhang, Y. (2014). A Simple Model for the Calculation of Plasma Impedance in Atmospheric Radio Frequency Discharges. *Plasma Science and Technology*, 924-929.
- GNURADIO.org. (2019). GNU Radio Conference 2019. www.GNURadio.org (p. 1). Huntsville, Alabama: GNU Radio.
- Goshvarpour, A. &. (2012). Radial Basis Function and K-Nearest Neighbor Classifiers for Studying Heart Rate Signals during Meditation. *International Journal of Modern Education and Computer Science*, 43-49.
- Graps, A. (1995). An Introduction to Wavelets. *IEEE Computational Science and Engineering Vol 2 Num 2*, 1-19.
- Guo, G., Wang, H., Bell, D., & Bi, Y. (2004). *KNN Model-Based Approach to Classification*. Belfast: Queen's University Belfast.
- H. R. Koenig, L. I. (1970 Mar). Application of RF Discharges to Sputtering. *IBM Journal of Reseach and Development Volume 14*, 168-171.
- H. S. Butler, G. S. (1963). Plasma Sheath Formation by Radio-Frequency Fields. *The Physics of Fluids* 6, 1346.
- H.Leyh, D. R. (1998). A new multi-term solution technique for the electron Boltzmann equation of weakly ionized steady-state plasmas. *Computer Physics Communications*.

- Hershkovitz, N. (1998 December). Role of Plasma-Aided Manufacturing in Semiconductor Fabrication. *IEEE TRANSACTIONS ON PLASMA SCIENCE*, VOL. 26, NO. 6, 1610-1620.
- Hiromichi Enami, Y. O. (2011). *History and Future of Hitachi's Plasma Etching System*. Kokubunji, Tokyo: Hitachi Rev.
- HiveMQ. (2018). *Introduction to MQTT*. Bavaria: HiveMQ.
- IBM. (2019). *Connect and protect with IBM MQ V9.1*. Rochester: IBM.
- J. Benedikt, D. E. (2011). *Diagnostics of low and atmospheric pressure plasmas by means of mass spectrometry*. Ruhr-University Bochum: Research Department Plasmas with Complex Interactions.
- J.P.Verboncoeur, M. C. (1993 february). Simultaneous Potential and Circuit Solution for 1D Bounded Plasma Particle Simulation Codes. *Journal of Computational Physics*, 321-328.
- James, R. D. (1973). Experimental Characterization of Microstrip Bends and Their Frequency Dependent Behaviour. *IEEE Confernece Digest* , 24-25.
- Jamimamul Bakas, M. H. (2016). A Comparative Study of Various Classifiers for Character Recognition on Multi-script Databases. *International Journal of Computer Applications (0975 - 8887)*, Volume 155, No 3, 1-5.
- Johnson, R. C., Ecker, A. H., & Hollis, S. J. (1973). Determination OF Far-Field Antenna Patterns From Near-Field Measurements. *Proceedings of the IEEE Vol 61 No 12 Dec* (pp. 1668-1694). Atlanta: IEEE-Xplore.
- Jon Tomas Gudmundsson, A. H. (2017). Foundations of DC plasma sources. *Plasma Sources Science and Technology*, No 26, 1-21.
- Jungwirth, P. (2016,Dec). *Power Spectral Density and Hilbert Transform*. Adelphi, MD: US Army Research Laboratory.
- Kelly, S., & McNally, P. (2017). Remote Sensing of a low pressure plasma in the radio near field. *Applied Physics Express* 10, 5.
- Kim, Y., Min, G., Kang, C., Cho, H., & Moon, J. (2005). Real-time process monitoring by optical emission spectroscopy in DRAM gate CD control. *ISSM 2005, IEEE International Symposium on Semiconductor Manufacturing*.
- Konstantin Y. Bliokh, P. S. (2012). *Electron Vortex Beams in a Magnetic Field: A New Twist on Landau Levels and Aharonov-Bohm States*. Maryland: American Physical Society.
- Kushner, M. (1988). A model for the discharge kinetics and plasma chemistry during plasma enhanced chemical vapor deposition of amorphous silicon. *Journal of Applied Physics*.
- Laguna, V., & de Andrade Lopes, A. (2010). *Combining Local and Global KNN with Cotraining*. Amsterdam: IOS Press.
- Langmuir, I., & Tonks, L. (1929). Oscillations in Ionized Gases. *American Physical Society*, 195-210.
- Learned, R., & Willsky, A. (1993). *A Wavelet Packet Approach to Transient Signal Classification*. Cambridge: Massachusetts Institute of Technology, Laboratory for Information and Decision Systems.
- Lee, G., Libedinsky, C., Guan, C., & So, R. (2017 May). Use of Wavelet Transform Coefficients for Spike Detection for a Robust Intracortical Brain Machine Interface. *8th International IEEE EMBS Conference on Neural Engineering* (pp. 540-543). Shanghai: IEEE.
- Lieberman, M. A. (1988). Analytical Solution for Capacitive RF Sheath 16. *IEEE Transactions on Plasma Science*, 638-644.

- Lieberman, M., & Lichtenberg, A. (2005). *Principles of Plasma Discharges and Materials Processing 2nd Edition*. Hoboken, NJ: Wiley.
- Lilly, J., & Olhede, S. (2011, Oct). On the Analytic Wavelet Transform. *IEEE Transactions on Information Theory*, 1-23.
- Liu, L. (2018 Mar 23). *Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL*. Ontario: University of Windsor.
- M A Lieberman, J. P. (2002 (June 14)). Standing wave and skin effects in large-area, high-frequency capacitive discharges. *Plasma Sources Science and Technology*, 283-293.
- M J Colgan, M. M. (1994). Very high-frequency capacitively coupled argon discharges. *Plasma Sources Science and Technology Volume 3 Number 2*, 181-189.
- M. A. Lieberman, A. J. (2005). *Principles of Plasma Discharges and Materials Processing*. Hoboken: Wiley & Sons.
- MacGearailt, N. (2018). *Process Diagnostics of Industrial Plasma Systems*. Dublin: Dublin City University.
- Mackus, A., Heil, S., Langereis, E., Knoops, H., & van de Sanden, M. (2010). Optical emission spectroscopy as a tool for studying, optimizing, and monitoring plasma-assisted atomic layer deposition processes. *Journal of Vacuum Science and Technology. A: Vacuum, Surfaces, and Films*, 77-87.
- Maeda, K. (2012). Study on the Distribution Control of Etching Rate and Critical Dimensions. *Japanese Journal of Applied Physics*, 51 (08HD01-1).
- Martinu, L. (2010). *Handbook of Deposition Technologies for Films and Coatings (3rd edition)*. New York: Elsevier.
- Maskrey, J., & Dugdale, R. (1966). The role of inclusions and surface contamination arc initiation at low pressures. *British Journal of Applied Physics*, 1025-1033.
- Matlab. (2018). *Mathworks Signal Hilbert Transform*. Natick, MA: MatLab.
- May, D. P. (1997, Jan 1). Plasma Diagnostics: Electron Temperatures and Ion Energy Distributions. *Online* (<http://www.chm.bris.ac.uk/~paulmay/misc/msc/msc4.htm>), pp. 1-4.
- Mehdi, R. (2016). Software Defect Prediction Using Radial Basis and Probabilistic Neural Networks. *International Journal of Computer Applications Technology and Research Vol5-Iss5*, 260-265.
- Mitchner, M. a. (1973). *Partially Ionized Gases*. New York: Wiley.
- Morgan, D. (2007). *A handbook for EMC Testing and Measurement*. London: The Institution of Engineering and Technology.
- Morton Mitchner, C. H. (1973). *Partially Ionized Gases*. New York: John Wiley and Sons.
- Mouli, C. (2005). Next generation manufacturing execution system (mes) enabling fully integrated fab automation in 300mm technology development and manufacturing. *IEEE Explore*, 11-12.
- Mulay, A. (2014). *SEMICON West 2014*. San Francisco: Semi.
- Mussenbrock, T., & Brinkman, R. P. (2006). Nonlinear electron resonance heating in capacitive radio frequency discharges. *Applied Physics Letters Vol 88 Issue 15*, 2-4.
- National Instruments. (2013, Sep 23). *Sensor Terminology*. National Instruments White Paper.

- O. Tarvainen, T. K. (2017). Electron cyclotron resonance ion sources – physics, technology and future challenges. *EPJ Web of Conferences* 149, 01003, 1-2.
- Pascal Chabert, N. B. (2011). *Physics of Radio-Frequency Plasmas*. Cambridge: Cambridge University press.
- Patrick McNally, S. K. (2018). *Radio Emission Spectroscopy (RES) for Industrial Plasmas*. Dublin: DCU.
- Paul A Miller, E. V. (2006). Spatial and frequency dependence of plasma currents in a 300mm capacitively coupled plasma reactor. *Institute of Physics Publishing : Plasma Sources Sci. Technol.* 15 (2006) 889–899, 889-899.
- Pedro Garcia Lopez, A. M. (2015).
<http://members.unine.ch/etienne.riviere/publications/LoMoEp-CCR-15.pdf>. CCR, 37-42.
- Pozar, D. M. (1998). *Microwave Engineering 2nd Edition*. New York: John Wiley & Sons.
- R Ediss. (2018). *Probing the Magnetic Field Probe*. Philips Semiconductor, UK: www.compliance-club.com.
- R.P. Brinkmann, R. F. (1996). A Reduced-Fluid Dynamic Discharge Model for Applications in Technology-Oriented Computer-Aided Design. *Journal of Electrochemical Society*, 1940-1944.
- R.W. Boswell, I. M. (1987). Self-consistent simulation of a parallel-plate rf discharge. *Applied Physics Letter*, 1-5.
- Rauscher, Christopher;. (2014). *Fundamentals of Spectrum Analysis*. Munich: Rohde & Schwarz.
- Reece Roth, J. (1995). *Industrial Plasma Engineering Vol1 Principles*. Bristol and Philadelphia: IOP.
- Reilly, M. P., Miley, G. H., & Lewis, W. (24-02-2009). Magnetic Field Probes for use in Radio Frequency Plasma. *Review of Scientific Instrument / Air Force Research Laboratory (AFMC)*, 14.
- Riley, M. E. (1995). *Unified Model of the rf Plasma Sheath (SAND95-0775, UC-401)*. Albuquerque, New Mexico: Sandia National Laboratories.
- Roy, K., & Roy, O. (2014, June). Performance Analysis of Optimum Receiver of Nakagami fading Channel for Non-coherent FSK System. *International Journal of Advanced Computer Research*, 527.
- Rudberg, E. (1930). *Characteristic Energy Losses of Electrons Scattered from Incandescent Solids*. Stockholm, Sweden: Royal Society Publishing.
- S. S. Harilal, M. S. (2004, Oct). *Laser plasma density measurements using interferometry*. San Diego: University of California, San Diego, USCD-ENG-114.
- S.B. Singh, N. C. (2008 September 28). Langmuir probe diagnostics of microwave electron cyclotron resonance (ECR) plasma. *Vacuum Volume 83 Issue 2*, 372-377.
- Sadiku, M. (1989/2007). *Elements of Electromagnetics*. Oxford: Oxford University Press.
- Saikia, P., Bhuyan, H., Escalona, M., Rawat, R. S., & Wyndham, E. (2018). A nonlinear global model of single frequency capacitively coupled plasma and its experimental validation. *AIP Advances Vol8 Issue 4*, 1-9.
- Sebastian Wilczek, J. T. (2016). Kinetic interpretation of resonance phenomena in low pressure capacitively couple radio frequency plasmas. *Physics of Plasmas* 23, 063514, 23.

- Semiconductor Equipment and Materials International. (2011). *SECS/GEMSEMI E30 1.3*. Milpitas, CA: SEMI.
- Severn, G. (2007/01/01). A note of the plasma sheath and the Bohm criterion. *American Journal of Physics - AMER J PHYS*, 1-8.
- Shi Jian Zhao, J. Z. (2004, 43). Monitoring of Processes with Multiple Operating Modes through Multiple Principle Component Analysis Models. *Industrial and Engineering Chemistry Research*, p7025-7035.
- Signal Hound Inc. (2016). *SA44B Specifications*. Battle Ground, WA: Signal Hound Corporation.
- SignalHound. (2017, Version 3, May 4th 2018). *SA44 and SA124 Application Programming Interface (API) Programmers Reference Manual*. La Center, WA: Signal Hound, Inc.
- Sk Park, D. E. (1990). Parametric study of a radio-frequency glow discharge using a continuum model. *Journal of Applied Physics*.
- Spikings, C.R.; Seddon, N.; Ibbotson, R.A.; Dolan, J.E. (2009). HPM Systems Based on NLTL Technologies. *IET Conference on High Power RF Technologies* (pp. 1-3). Bristol: BAE Systems.
- Sternberg, N., & Godyak, V. (2007, Oct). The Bohm Plasma-Sheath Model and the Bohm Criterion Revisited. *IEEE TRANSACTIONS ON PLASMA SCIENCE* No. 5, p. 10.
- Steven, J. E., Sowa, M. J., & Cecchi, J. L. (1996-01-01). Uniformity of radio frequency bias voltages along conducting surfaces in a plasma. *OSTI.GOV*.
- T. H. Chang, L. R. (1999). *Dual-function circular polarization converter for microwave/plasma processing systems*. Maryland: American Institute of Physics.
- Tektronix Inc. (2019). *RSA306 Spectrum Analyzer specifications*. Beaverton, OR: Tektronix Inc.
- Ulrich, T. J. (2006, March, Los Alamos National Laboratory). Envelope Calculation from the Hilbert Transform. *Researchgate Publication 257547765*, 1-6.
- University Lowell Massachusetts . (2019). *Windows and Leakage Brief Overview, Dynamic System, Rev 070606*. Lowell: UMASS, Lowell.
- V. Baglin, J. B. (2000). The Secondary Electron Yield of Technical Materials and its Variation with Surface Treatments. *Proceedings of EPAC 2000, Vienna, Austria*, (pp. 217-221). Vienna, Austria.
- V.M. Borisov, . (2014, Nov 1). EUV light source with high brightness at 13.5 nm. *Quantum Electronics* 44, pp. 1077-1083.
- Vaclav Satek, P. V. (2019 (Sept)). Taylor Series Based Integration in Electric Circuits Simulations. *Mathematical Analysis and Numerical Mathematics*, 352 - 359.

Appendix A Microwave Plasma Excitation with Autotuner

A.1 Introduction

Low pressure processing plasmas are widely used in the manufacture of semiconductor devices (Commission on Physical Sciences, 1991). Much of today's deep silicon etching is performed by means of the Bosch Process (Dixon-Warren, 2018, May 25). The basic process involves the repeated cycle of alternating etch and deposition chemistries. For the semiconductor industry, plasmas are principally distinguished by the method by which they are powered: those where charged particles are accelerated in a sheath region next to the electrodes and those where charged particles are accelerated in an electrical field presented in the volume (Hershkowitz, 1998 December). An example of the former is the capacitively powered plasma, though there are some multi-frequency capacitively coupled and high voltage DC powered types (Jon Tomas Gudmundsson, 2017). The latter type includes a variety of resonance discharges such as the microwave (electron cyclotron resonance, ECR) and inductively coupled RF discharges (M. A. Lieberman, 2005). There are some common questions for plasma processes:-

- Can the process be accurately predicted?
- Can anomalies be detected?

These questions go to the heart of ensuring a repeatable, production quality manufacturing process. In this Appendix we examine the extraction of data from an autotuner subsystem as applied to the Hitachi 712 ECR Coupled plasma chamber. Critical control of the parameters of the plasma is of vital importance and a number of measurement techniques have been developed. These include invasive and non-invasive methods. The plasma excitation frequency spans from DC to microwave. In the particular case of the Hitachi 712 chambers, the plasma is excited by a microwave source operating at 2.4GHz into a vacuum chamber with a ~ 0.850 tesla magnetic field. This is the Electron Cyclotron Resonance (ECR) excitation of the plasma. The source (microwave magnetron via waveguide) and destination impedance are normally matched by a set of fixed tuning stubs, but may be replaced by a Daihen SMA-20B microwave stub tuner. This unit permits the tuning to be performed manually or automatically, but in normal operation it

is in standalone automatic mode. One of the main challenges for these systems is to match chamber and plasma characteristics to the autotuning parameter outputs and to build a matching model.

For these systems, it is usual to have a fixed 'recipe' for the operation of the process. The recipe-known power is applied, the chuck bias voltage and RF excitation is applied, and the recipe-known flow and combination of gasses at the correct pressure is injected into the chamber. The magnetic containment is controlled via a programmable parameter-based power-supply (of the order of ~ 290 V dc at ~ 3 A for the Hitachi 712) until the process is complete. For the Hitachi 712, there is an Optical Spectrometer (Ocean Optics HR2000) located on the side of the containment vessel which is utilized for determination of the endpoint for some of the operations. Other plasma sensors which can be used include Langmuir probes (S.B. Singh, 2008 September 28), laser interferometry (S. S. Harilal, 2004, Oct), and mass spectrometry (J. Benedikt, 2011). On some machines there may also be exhaust gas analysis at the turbo pump outlet. These require a vacuum port and may interfere with the plasma and require some engineering for the tool. A physical sensor in the chamber may also be subject to the plasma, may interfere with it and may introduce contaminants into the chamber.

The ranges of power used in industrial plasmas, even in the semiconductor industry is quite varied with similar machines operating from 70KW to tens of watts. In all, this makes the design and application of sensors a very challenging area for the monitoring of plasma systems. The particular Hitachi 712 described in this report operates with magnetron power in the 1 kW range. We examine methods to monitor the activity of the plasma and production quality via a number and variety of measurements, with the main thrust being sensors of the non-invasive type.

A.2 ECR Powered Plasma

In this more straightforward mechanism, we monitor the control of the capacitive effect of the stub control into the microwave waveguide to respond to the variation of the perceived electrical load of the plasma as the recipe is processed.

This is a non-invasive method of monitoring a parameter of the plasma. In the particular case of the Hitachi 712 chamber, the plasma is excited by a microwave source operating at 2.45GHz (also monitored by RF spectral analysis) into a vacuum chamber with a ~ 0.0850 tesla magnetic field (see Figure 127 below). The source and load impedances are normally matched by a set of fixed tuner stubs (set at nominal operating range), but may be replaced by a Daihen microwave stub tuner. This permits the tuning to be performed automatically by monitoring the reflected power from the plasma.

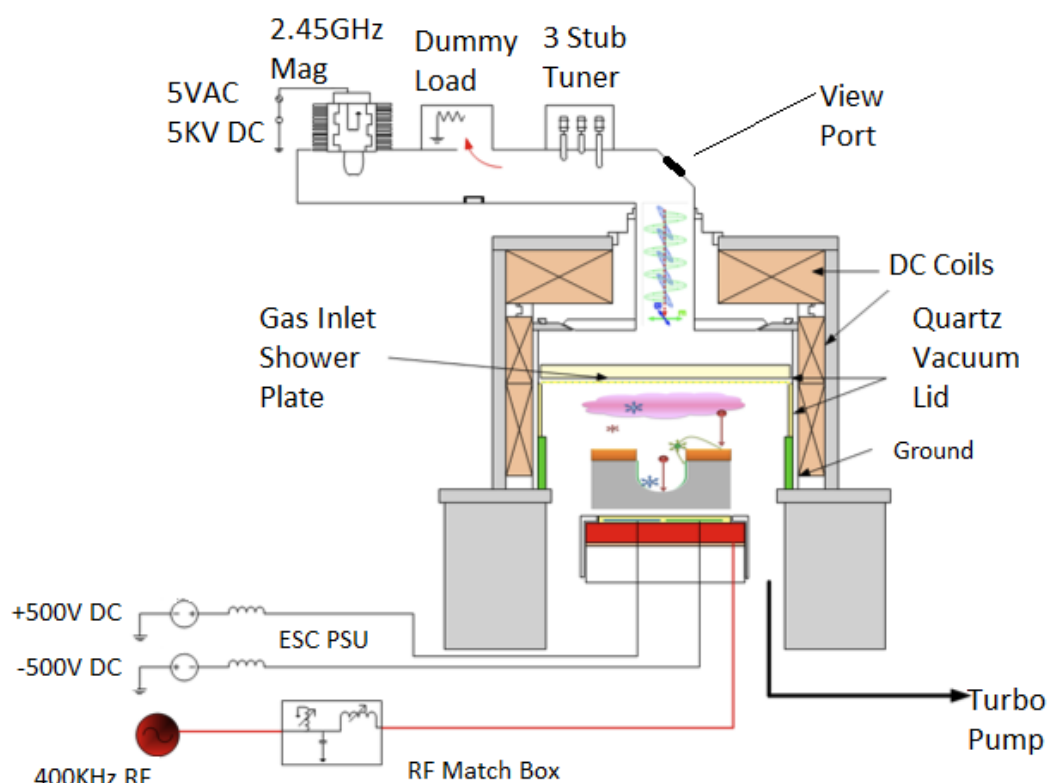


Figure 127 Hitachi 712 ECR Plasma with Daihen Stub Autotuner

The figure (Figure 127) above shows the main components of the Hitachi 712 ECR etch tool. The microwave energy is generated by the magnetron, passes through a duplexer and on to the Daihen three stub autotuner. The microwave energy then passes into the chamber to excite the plasma through the quartz window which maintains the vacuum.

A.2.1 Daihen Microwave Autotuner

The autotuner replaces the static 'bolt' type tuner, which takes no account of the varying load impedance presented by the plasma as it operates (393, 1970).

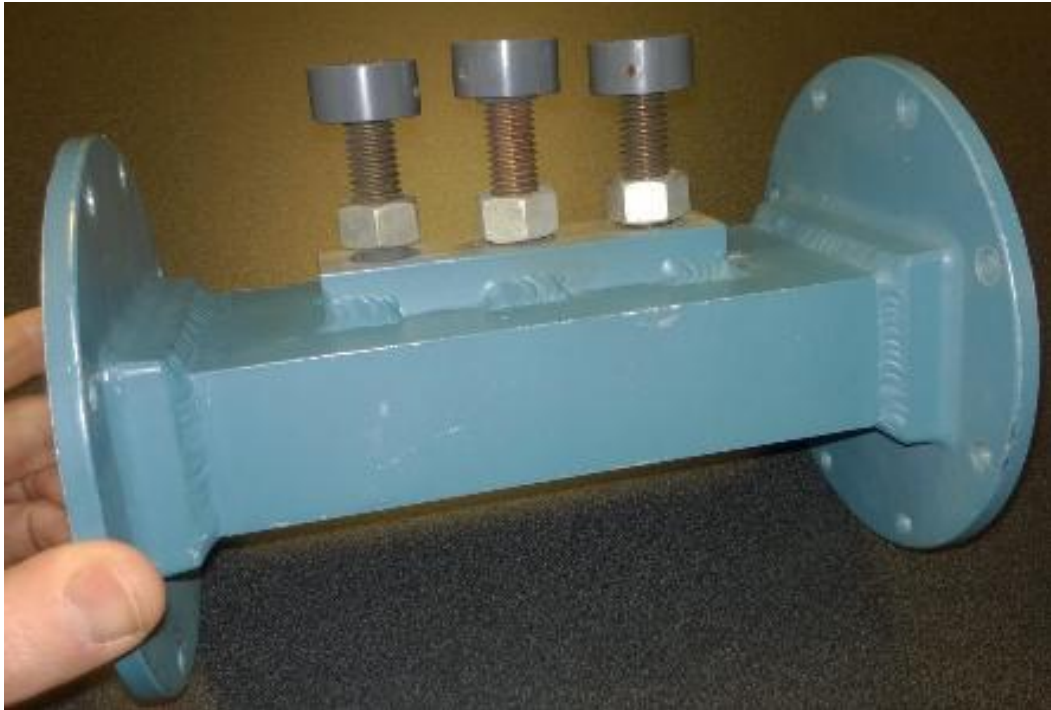


Figure 128 Example 3 stub manual tuner

The figure (Figure 128) shows an example 3-stub manual microwave tuner. Note that this is not the typical WR340 waveguide as used on the Hitachi 712 tool and is used here to merely show the relatively crude mechanism of the tuner. For the Hitachi 712 tool, during machine maintenance, a recipe is run on the machine that attempts to represent the normal wafer processing operation of the machine. With the machine operating and the magnetron at nominal power setting, the three M10 bolts are adjusted for minimum reflected power as measured at the return port of the duplexer.

The microwave generation of plasmas depends on the high electric fields generated by the magnetrons in the low-pressure gas. With a steady B-field, there is resonance between the applied frequency ω and the electron cyclotron frequency ω_{ce} .

The electron cyclotron frequency or electron gyration frequency (O. Tarvainen, 2017) is given by:-

$$\omega_{ce} = \frac{eB}{m} \text{ (or } f_{ce} = 2.8 * 10^4 B \text{ MHz)}$$

With the applied magnetic field of ~ 0.0875 T this allows a common 2.45GHz magnetron to be used to energize the plasma. The typical system which is utilized in the Hitachi 712 is a commercial (though carefully selected for frequency and harmonics) magnetron that is fed into a duplexer / 3 port circulator, with the return port connected to a dummy load. The output port of the duplexer / 3 port circulator is fed to the Daihen autotuner (see section below) and then through a wave guide to the mode converter and into the magnetic containment through the quartz window where the plasma will be formed over the wafer and back gas cooled RF chuck. The energy flow diagram is shown in the figure below (Figure 129).

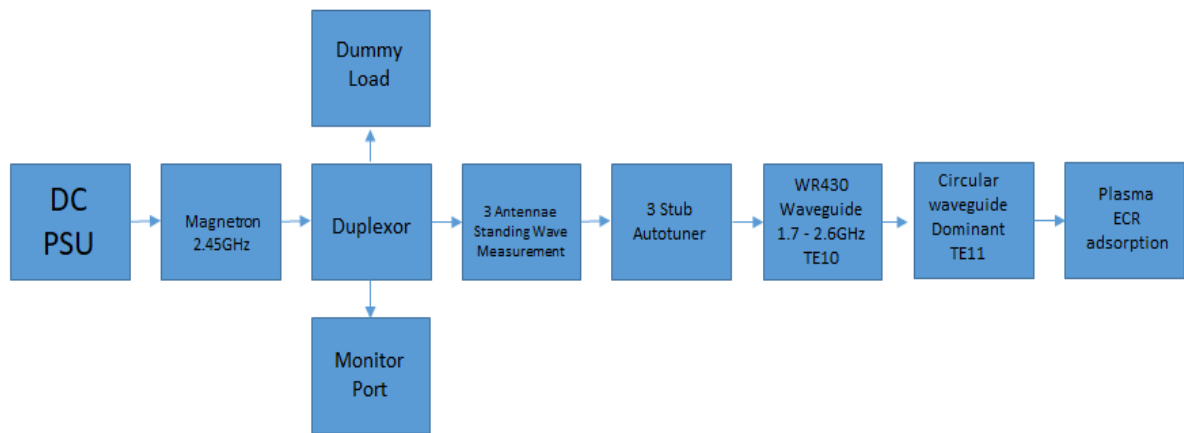


Figure 129 Electron Cyclotron Hitachi 712 Energy Flow.

The autotuner replaces the static ‘bolt’ type tuner, which takes no account of the varying load impedance presented by the plasma as it operates. From the initial strike, to the variation in the gas mix, the impedance goes through a number of distinct phases and these can be tracked through the monitoring of the operation of the autotuner. The autotuner matches the impedance of the magnetron to the load impedance of the plasma. There are a wide variety of components and circuits

for impedance matching, but basically the operation is to match the impedance to maximize the transfer of power from the source to the load. A plot of load power versus load impedance shows characteristic maxima at the 50% power point where $Z_{Load} = Z_{Source}$. The generator has to be able to generate at least twice the power that is required to be delivered into the load at optimal matching. This can be seen clearly in the figure below (Figure 130) for power transfer vs load match plot. By calculating and then plotting the transferred power for a varying load impedance, the following plot is obtained.

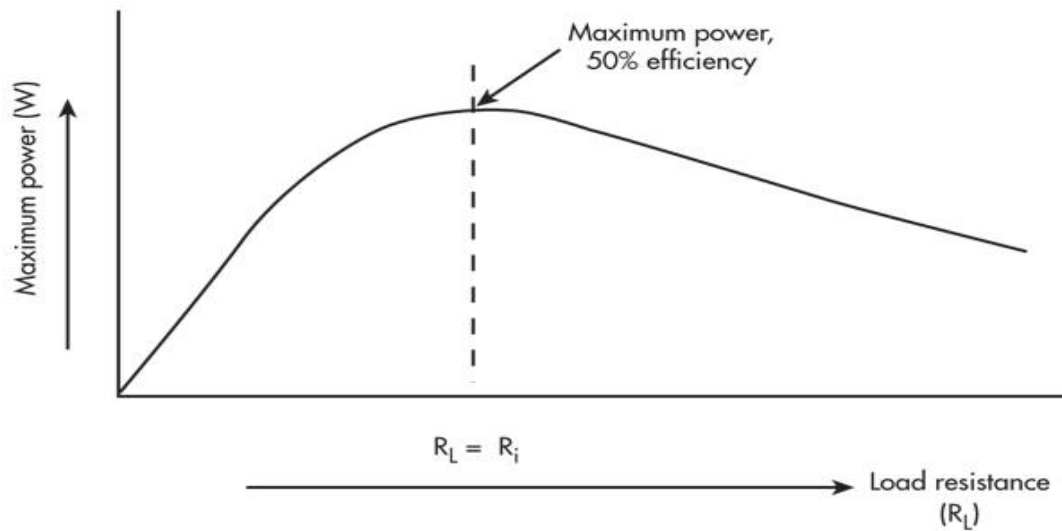


Figure 130 Power Transfer for Source vs Load Impedance

The microwave guide for the Hitachi tool is WRJ2 (JIS) 2:1 rectangular section comprises two impedances:

For Transverse Electric waves.

$$Z_{TE} = \frac{Z_o}{\sqrt{1 - \left(\frac{f_c}{f}\right)^2}}$$

For Transverse Magnetic waves.

$$Z_{TM} = Z_o \cdot \sqrt{1 - \left(\frac{f_c}{f}\right)^2}$$

where Z_o is the impedance in free space ($\sim 377\Omega$) and f_c is the cutoff frequency of the waveguide for the particular operating mode (see following). This is an expression of the relationship between electric-field and magnetic-field intensities in an electromagnetic field propagating through a vacuum. The Z_o of free space is intrinsic and is generally expressed in ohms (Ω) and is theoretically independent of wavelength in dry air (excessive humidity or high concentrations of particulates reduces Z_o).

However, generally considering waveguide impedance as a true impedance is quite difficult. The impedance is of the order of 500Ω with a downward slope. As the cutoff is reached, wave impedance tends to infinity.

There are several ways to define the waveguide impedance (Pozar, 1998):

- a. Use the voltage measurement between the middle and the top and bottom of the walls combined with the value of current to be integrated across the top wall.
- b. Determine waveguide impedance by measuring voltage and power flow within the waveguide.
- c. Determine the ratio of electric field to magnetic field at the centre of the waveguide.

All these methods tend to give results within a factor of two of the free-space impedance of 377Ω .

The waveguide post or screw is made from a conductive material (aluminium in the case of the Daihen SMA-20B). The element is capacitive by only extending partway through the top wall. The bend (mitre) in the waveguide which is used to direct the microwave energy into the plasma is an E-field bend with two 45° bends (the location of the camera port). Effectively the signal is reflected, and using the 45° surface, the reflections occur in such a way that the fields are left undisturbed, although the phase is inverted. The top surface of the bend (camera mount) should be of the order $\frac{\lambda}{4}$ in length.

The Daihen SMA-20B has three tuning screws (operated by stepper motors) that are placed approximately $\frac{3\lambda}{8}$ apart. For 2.45GHz the wavelength is 12.2cm and thus the distance between each tuning screw should be 4.57cm and whose penetration into the waveguide can be varied. In the case of the Daihen SMA-20B, it has its three tuner stubs closer together at 3.75cm between centres. The value of susceptance increases with diameter of the post and also increases with depth, which can be attributed to the increase in capacitance between the tip of the post and the bottom surface of the waveguide. At a particular depth, the post becomes series resonant and is inductive for greater depths up to and including maximum insertion (Pozar, 1998). Below is a block diagram of the autotuner section. With the input from the left side, the three antennae measure the standing wave and attempt to reduce that by moving the three adjustable stubs.

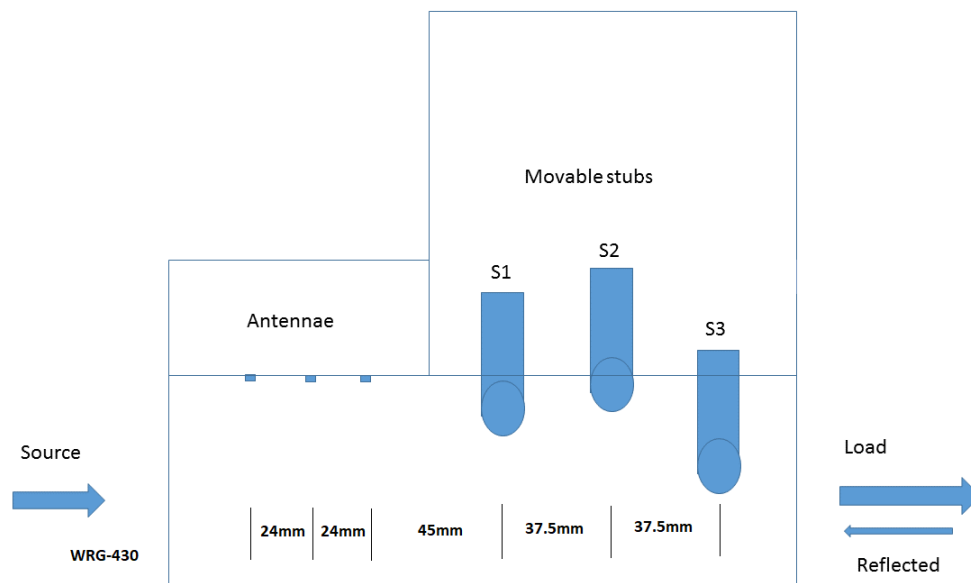


Figure 131 Daihen SMA-20B Autotuner Mechanics

The rectangular waveguide (WR430) has a number of operating modes. The cutoff frequency f_c is determined for a particular mode by the dimensions of the waveguide (Sadiku, 1989/2007). The dominant mode is the one where the cutoff frequency is the lowest.

The cutoff frequency can be determined by

$$f_{c_{mn}} = \frac{c}{2} \sqrt{\left(\frac{m}{a}\right)^2 + \left(\frac{n}{b}\right)^2}$$

where a = width, b = height, c = 3 x 10⁸ m/s (for air), m=0,1,2,3... and n=0,1,2,3...

Calculating the cutoff frequencies for the first 4 modes for the WR430 with internal dimension of 109.2 mm x 54.6 mm gives:

Mode	f_c (mn)
TE10	1.373 GHz
TE01	2.747 GHz
TE11	3.071 GHz
TM11	3.071 GHz
TE20	2.747 GHz
TE21	3.885 GHz

Table 9 Frequency Cutoff for Differing Modes of the WR430 Rectangular Waveguide

With the source being a magnetron at 2.45 GHz, the dominant mode will fall to TE10 in the waveguide as TE10 is the only one supported, the rest being in cutoff mode. In contrast to the rectangular waveguide, the dominant mode in a circular waveguide is TE11. The signal is then converted using a mitred right-angle mode converter from TE10 to TE11 circular at 110 mm diameter (James, 1973). There is shunt capacitance due to the bend, but this may be compensated by the preceding tuner. Using just a rectangle feeding into a cylinder leads to peaking of power flux on the axis. This can cause a non-uniform plasma generation and hence asymmetric wafer processing. (As part of this investigation, we planned to mount a camera axial to the wave guide and using selected optical filters to view the plasma in 2D mode. We had hoped to synchronize the camera frame acquisition to the RF excitation of the chuck. However, because of resource constraints, this remains for further research). The optimum is to introduce a circularly polarized wave as the gyrating electrons rotate in phase with the right-hand circularly (RHP)

wave, seeing a steady electric field. The behaviour of the electrons is a little more complicated than would be first considered. It might be thought that they should rotate with a single frequency, but experiments in Japan and Belgium have shown that they exist at a multitude of frequencies. These are namely, the cyclotron frequency, zero frequency and the Larmor frequency (half the cyclotron frequency) (Konstantin Y. Bliokh, 2012). For the Hitachi system, a wave of arbitrary polarization launched into the chamber can be viewed as the sum of two counter-rotating circular polarized waves. One of the components which rotates in the same sense as the electrons will resonantly react with them. These electrons will continuously gain transverse energy but they rotate with a spread of frequencies. Researchers have reported on orchestrating the impinging wave to be circularly polarized so as to azimuthally improve the plasma (Maeda, 2012) (T. H. Chang, 1999).

A.1.2.2 Daihen Automatic Microwave Tuner

Plasmas with very high densities 10^8 to 10^{12} electrons cm^{-3} are prevalent in the micro-electronics industry and whilst the parallel plate plasma systems are predominant, high quality microwave generated plasmas in magnetic confinement are becoming more commonplace (Hiromichi Enami, 2011). This has been the result of continual development of the associated microwave components and power-supplies. The Hitachi ECR unit utilizes a commercial magnetron followed by a guidewave duplexor and then a Daihen Autotuner before directing the microwave energy into the magnetically confined plasma chamber. A test system was designed and built to acquire data from the autotuner and map this into a database system as part of the multi-variate data acquisition of the plasma process as wafers were being processed.

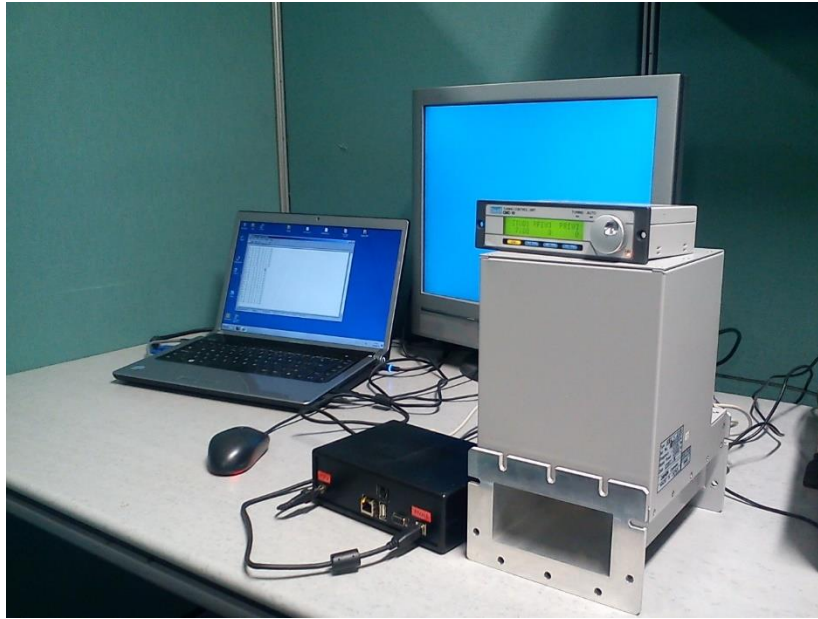


Figure 132 Daihen WR430 Autotuner, Controller and Acquisition Box

There are two distinct components to the Daihen autotuner. The first component is the CMC-10A autotuner controller and the second part is the WG-430 waveguide component with antenna pickups and stepper motors to insert/remove stubs from the waveguide section. Bespoke acquisition software has been written to operate on a PC (and embedded device as shown in the picture above) to collect the stub positions from the CMC-10A autotuner controller. This code is included in the Appendix M. The CMC-10A controller operates the SMA-20B stub autotuner attached inline with the waveguide section inserted between the duplexor and the load. The software acquires the power, forward and reflected, U_{load} and V_{load} and the U_{in} and V_{in} values.

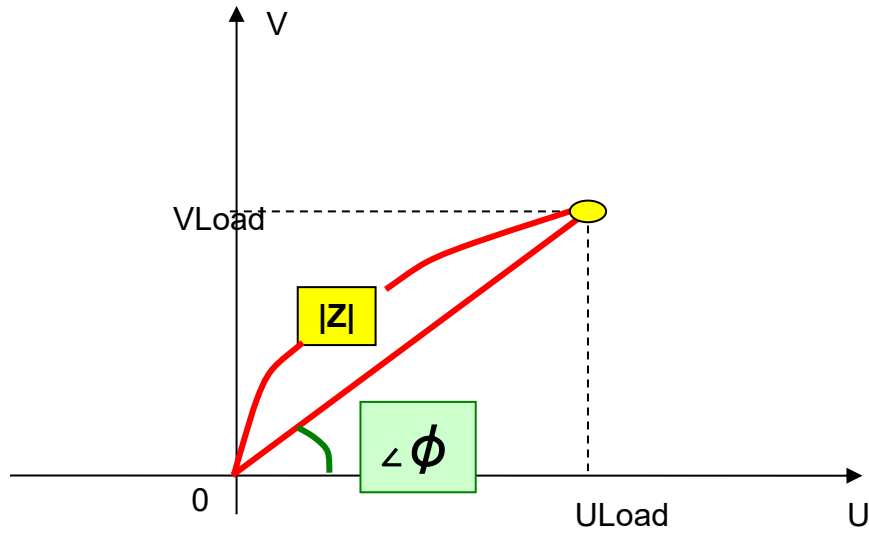


Figure 133 Mapping of the Voltage to Current at Load

The plotting of the current at the load to the voltage at the load gives rise to the impedance.

The magnitude and phase relationship is given by the following:-

$$Z = \sqrt{U^2 + V^2}$$

$$\phi = \tan^{-1}\left(\frac{V}{U}\right)$$

For the CMC-10A autotuner values, these values are recorded at 2Hz and are then transferred via the (previously described) Message Queueing databus and stored in a database. The aluminium stubs each with a diameter of 11.23 mm are spaced at 37.5 mm centres and are located downstream of the antennae which are 45 mm ahead. The three antennae are located on 24 mm centres (See Figure 133 above). This collection of data for the autotuner is an example of Virtual Metrology and is used in the semiconductor industry in a significant manner. Physical and electrical parameters of devices are estimated from the information collected from the various sensors located on the tools and from other non-direct sources. Whilst this data is not a direct measure of parameters, as side-channel data it can be very accurate (Ariane Ferreira, 2009 June).

A two-step etch process of ($SF_6 / Cl_2, NF_3$) is run on the Hitachi 712 ECR chamber and the data from the Daihen SMA-20B/ CMC-10A autotuner is recorded. (MacGearailt, 2018). The recipe is used to etch under the transistor gate. This is a proprietary confidential configuration. The timings of the etch process, the gate dimensions and exact chemistries are not released.

1. The first step is to introduce a dilute SF_6 and helium mix to etch through the native oxide film on the surface of the wafer and to etch down vertically to a targeted depth. To minimize damage to the surface, the ion flux must be reduced to the lowest level whilst still performing the function. The chuck active bias is off at this stage.
2. With the plasma power still being applied, the chemistry is changed with the mass controllers to a $Cl_2 / NF_3 / Ar$ chemistry. The pressure is increased and the microwave power is reduced. This creates an isotropic etch which undercuts the gate.

We do not have the exact timing of the chemistry introduction to the chamber and thus the corresponding plasma reaction.

Two sample runs are shown, where the tuner is compensating for the plasma impedance. These are plotted below.

The sample run acquisition is included in the Appendix N. As per the Daihen manual (Daihen Corporation, 1995), the values given for the stub positions are floating values in the range 0.00 to 25.00. These are not millimeters and are described in the manual for the autotuner as arbitrary units. The time range on the scale is from 41:09.0 (m:s:ms) to 43:42.5. The device is polled through an asynchronous serial line approximately every 0.5 second (the time dependency is a function of the message length returned and the response time of the microprocessor to the polling requests). The start time in seconds is 2,469 seconds and runs to 2,622 seconds with approximately two samples per second giving a run of just over 320 samples for 153 seconds as graphed below:

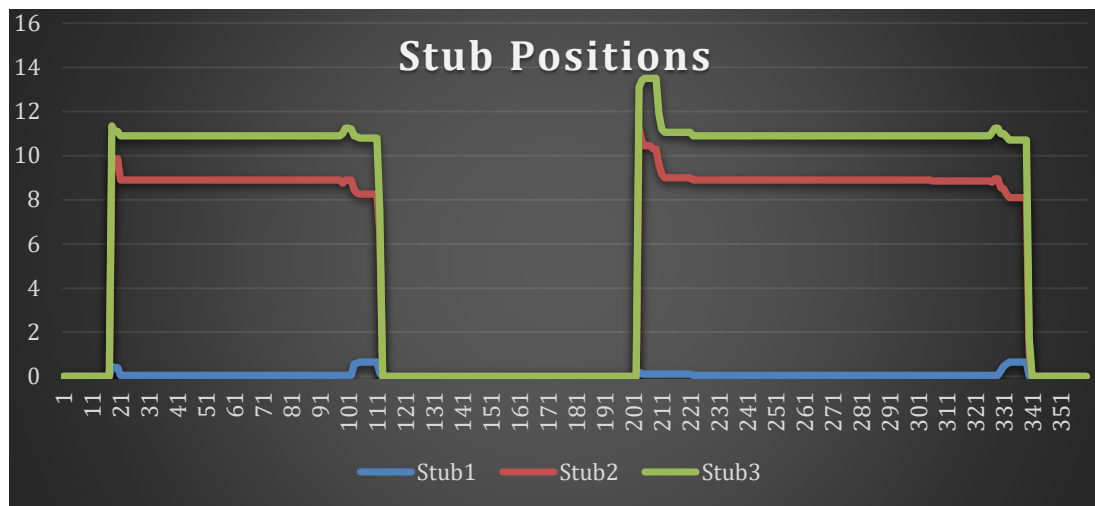


Figure 134 Run 1, Stub Recorded positions for Two-step Wafer Etch Recipe run

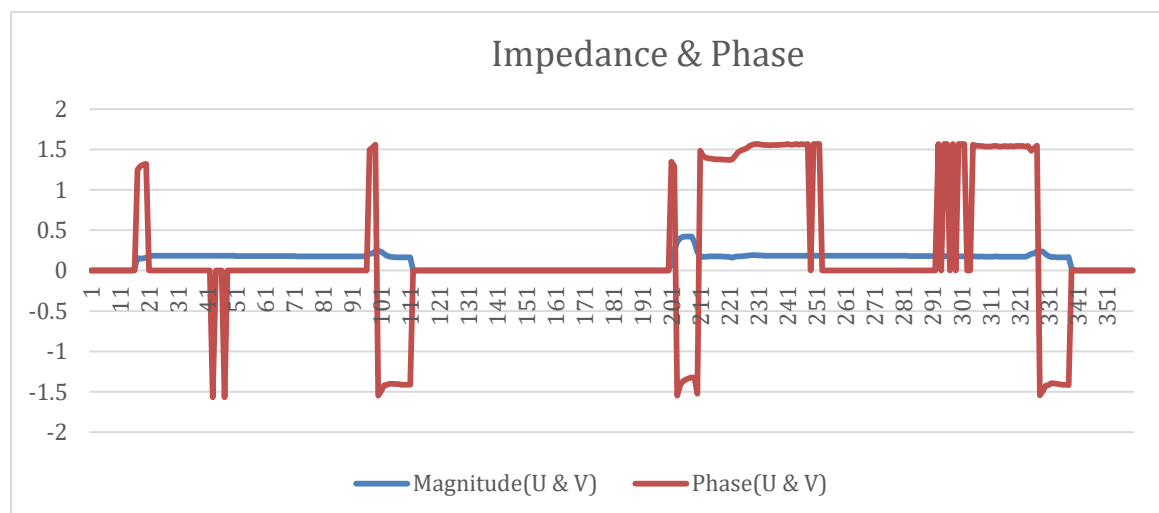


Figure 135 Run 1, Impedance and Phase Correction for Two-step Wafer Etch Recipe Run

From the figure above (Figure 135) we see very little change in the real impedance (Magnitude (U & V)) of the plasma but a high rate change in the susceptance (imaginary part, Phase (U & V)). The stubs then repond to correct this phase change. As per NBS Technical Note 393 re the Design on Multi-screw Waveguide Tuners (393, 1970) stubs are placed based upon three simplifying assumptions:-

1. *Dominant mode propagation exists in a waveguide everywhere except in the near vicinity of the tuning screws. That is, higher order waves set up by one tuning screw are sufficiently attenuated before reaching another screw.*
2. *The multi-screw tuner is lossless. This is a good approximation for analytical purposes.*
3. *Reflection coefficients to be realized at the input port are small in magnitude (less than 0.5).*

Note:- The model of the cylindrical stub to be used in this analysis is that of shunting capacitive susceptance.

The model constraints imply that only admittances inside the $|r| = 0.5$ circle can be realized. Thus as each stub is inserted into the waveguide (before it becomes series resonant) it approximates a shunt capacitance. Series resonance should not happen on the Daihen SMB-20A as the insertion depth is less than the critical quarter wavelength insertion depth of $12.2\text{cm} / 4$ giving a depth of 3.05cm . Thus stub (screw) 1 & 2 can move the wave towards the generator (120°), stub 2 & 3 can move it away from the generator (120° to 240°) and stub 1 & 3 can move the wave from 240° to 0° .

This can be shown on the normalized Smith admittance chart below:-

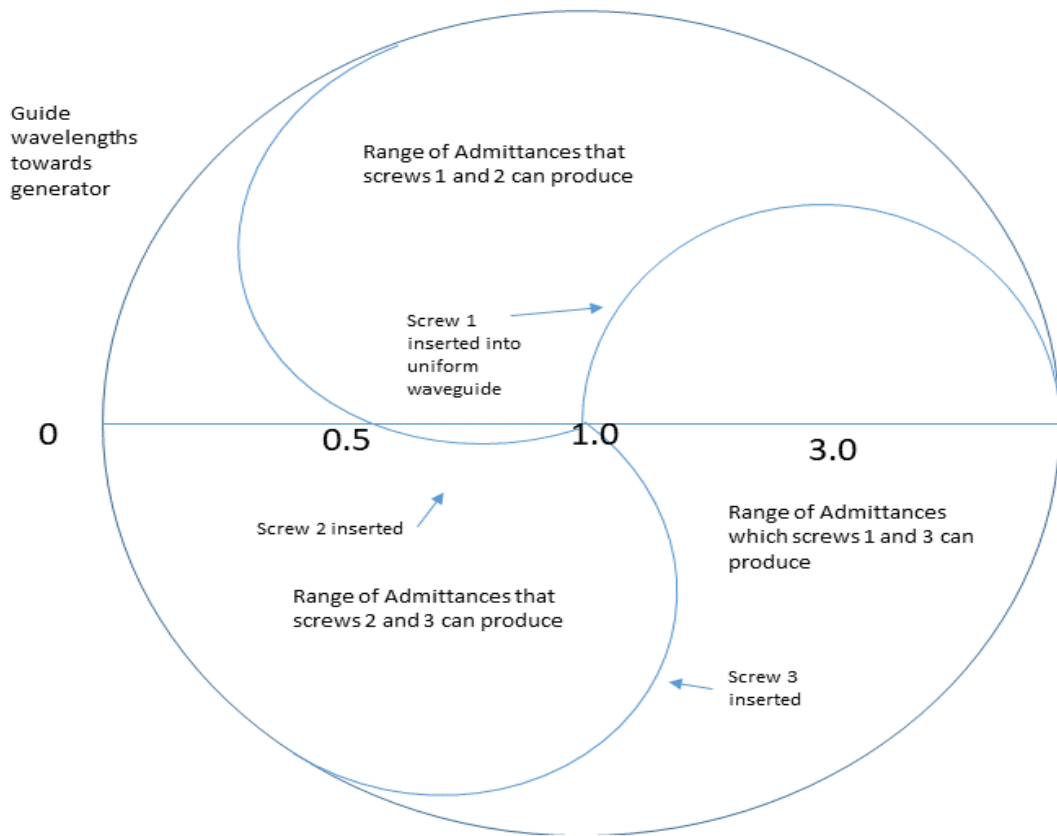


Figure 136 Normalized Admittance Smith Chart Stub Operation for 3 Stubs

From these phasors recorded through the polling software and the graphing above we can see the capacitive changes in the plasma. While the autotuner controller does not compensate once the reflective power magnitude stays within the 0.5 band (or as set by the threshold level in its programming), the phase change reported indicates rapid plasma changes in operation. This gives a V_{load} matching (indicating how much of the incident wave is being reflected) as shown below:

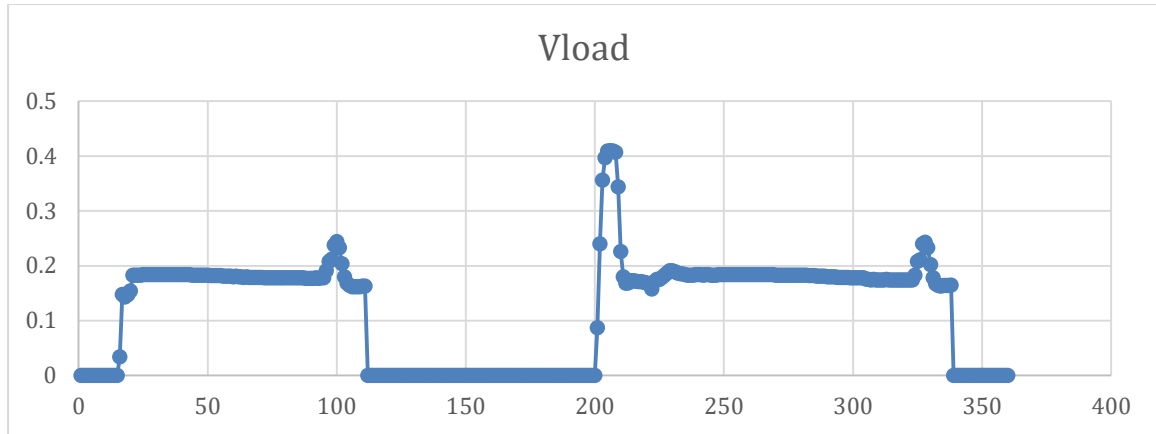


Figure 137 Run 1, V_{load} Reflection Coefficient Matching on Daihen Autotuner

The corresponding U_{load} graph is as follows:

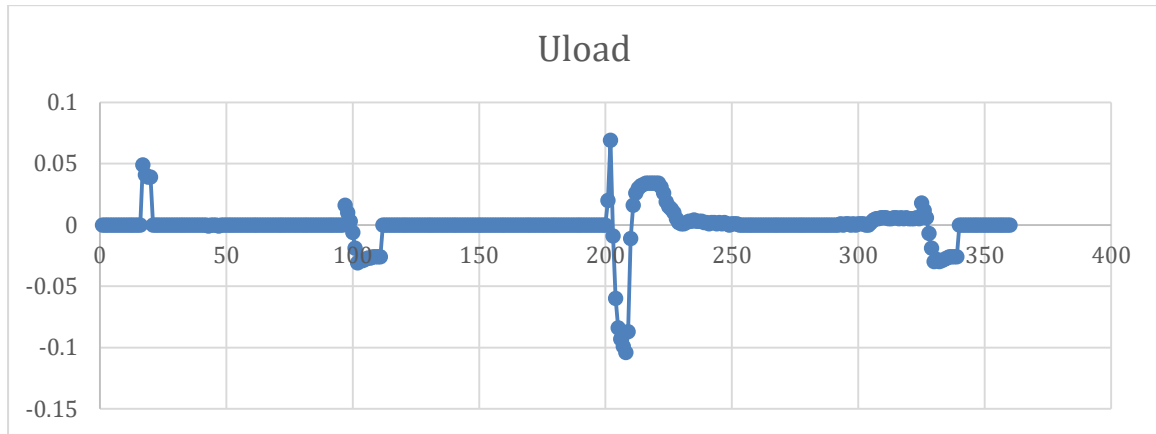


Figure 138 Run 1, U_{load} Reflection Coefficient Matching on Daihen Autotuner

The stubs, when inserted into the waveguide interrupt the electric field and create a capacitance and increase the reflection. Each stub being in a different location along the waveguide, can affect the phase of the reflection. With the combination of stubs it is possible to modify the reflected wave to appear resonant in the load and thus the transmitter (magnetron) sees maximum power transfer. When plotted on the Smith chart we can see the stub 1 correction for admittance with the insertion of the stub into the waveguide.

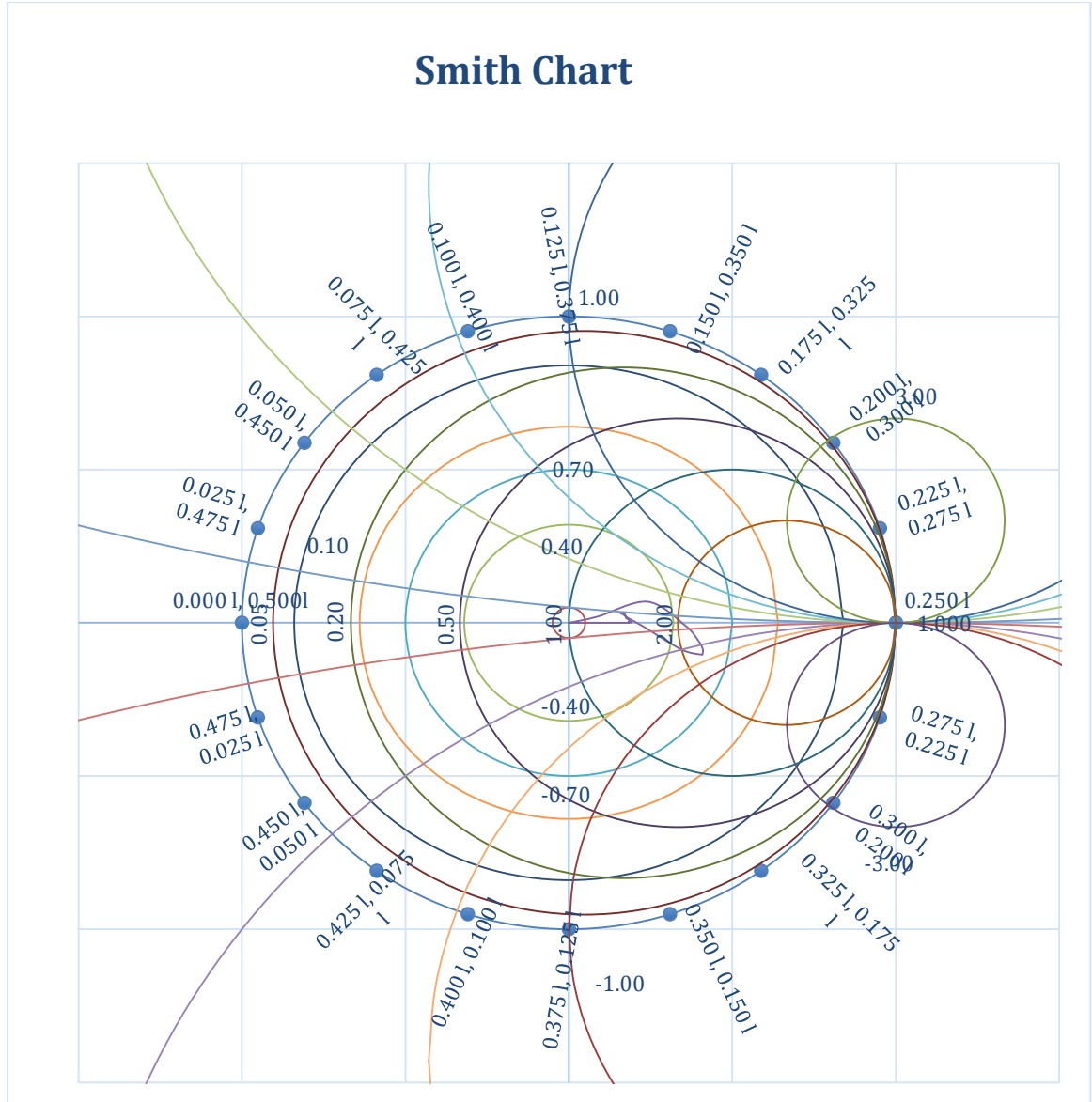


Figure 139 Smith Chart Plot of Daihen Autotuner Correction

We note that the impedance is quite closely matched to that of the load (centre 0,0, $z = 1$) and stays within the circle described by $|r| \leq 0.5$. This is a limit of the tuning for the Daihen autotuner and as approaches $|r| \sim 0$ it does not make stub corrections as this can lead to oscillations. The antennas are measured every $500\mu\text{s}$ and this is then averaged over a period to 0 to 1000 samples. There is hysteresis for the V_{load} value with a threshold level set by the user. This nominally defaults to 0.5 and the U_{load} target default value is 0° . For a uniform transmission line (and the waveguide with its mitre joint and the mode-converter and the plasma do not represent such), the complex coefficient of a standing wave varies

according to the position on the line. For a lossless transmission line, this is represented by a circle centred at 0,0 with phase changing as we progress around the line. The phase represents the wave along the line. For 2.45GHz, the wavelength is 12.2 cm. With a constant state of plasma conditions, the position on the Smith chart represents the standing wave to the plasma absorption. The mitre joint with its capacitive effect will need to be taken into account. It was not possible to modify the position of the 0.0875 T line to move the absorption location of the microwave energy. This is for future examination.

For the same recipe and another run, the following plots were obtained. In this case, we were able to record the forward power as applied to the autotuner by the magnetron source as controlled by the static recipe.

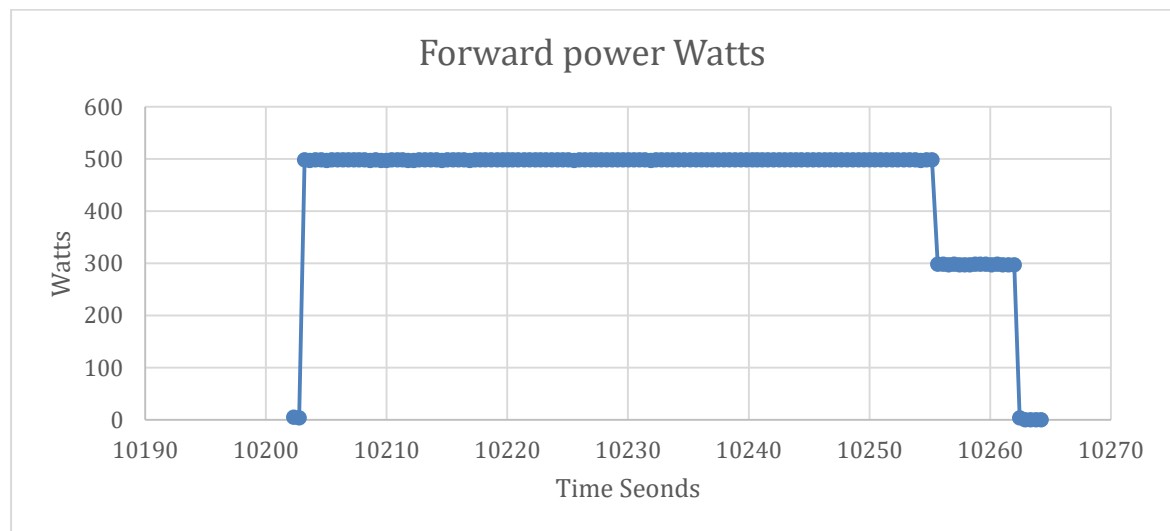


Figure 140 Run 2, Forward Power as Measured by Daihen Autotuner Antenna

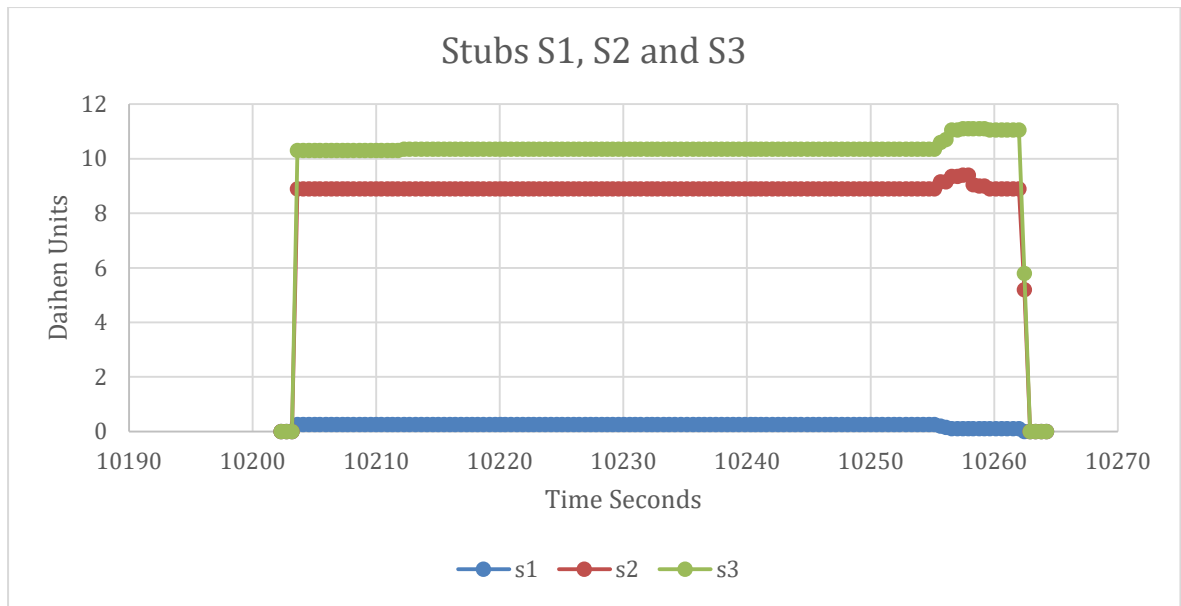


Figure 141 Run 2, Daihen Stub 1, 2 and 3 Recorded Positions (into the Waveguide)

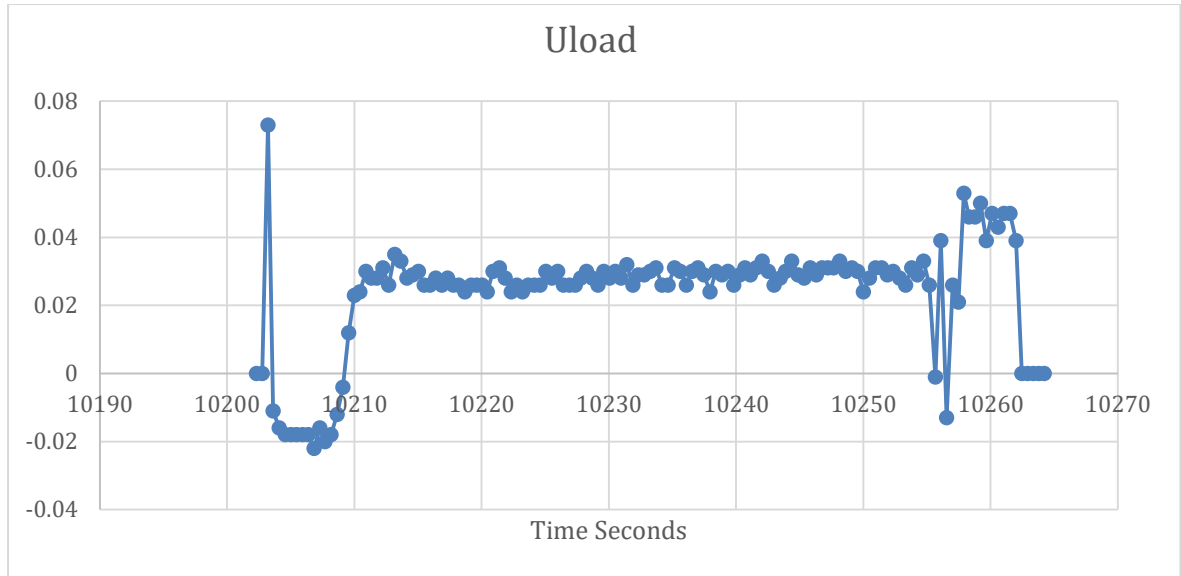


Figure 142 Run 2, The Corresponding U_{load} Matching by the Daihen Autotuner

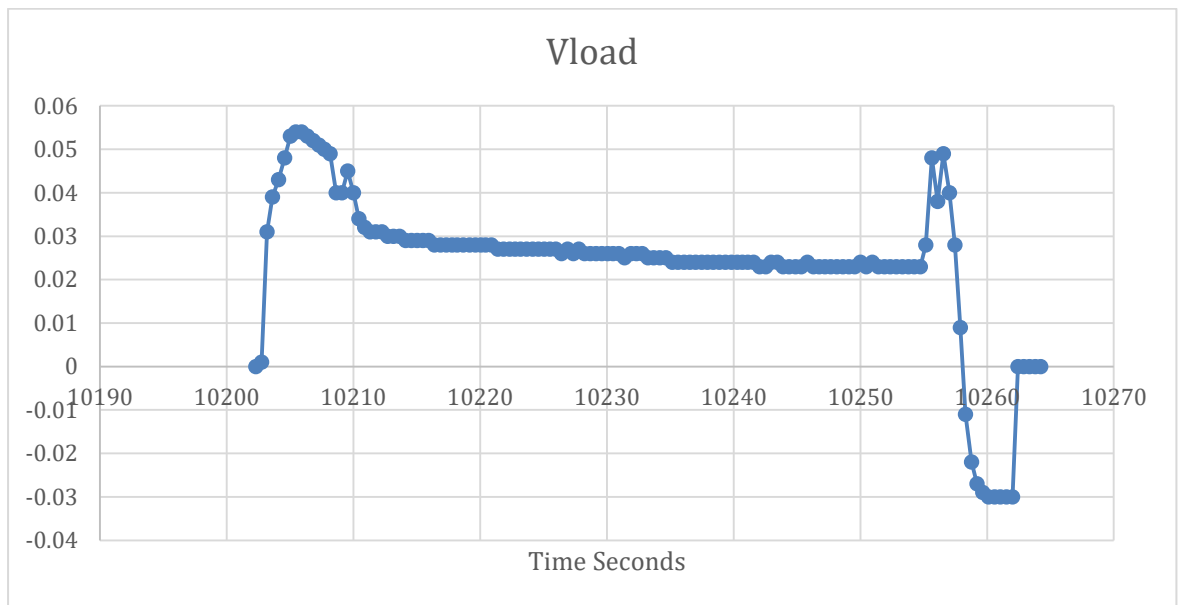


Figure 143 Run2, V_{load} as Recorded from the Daihen Autotuner

We have determined that the autotuner makes significant changes to the wavefront as presented to the plasma and it does this by adjusting the stubs/screws to adjust for the change in capacitance in the plasma. It was not possible to continue this investigation due to access to the equipment. A complete magnetron, duplexer and autotuner waveguide was obtained to be installed on the DCU Hitachi 712 plasma chamber. This would permit further investigation to be

undertaken. It was envisaged that we would perform modelling with time-varying sheath model and mapping to bulk impedance with autotuner stub matching recordings. This would permit equivalent circuit modelling, impedance/susceptance correction phase, and the observation of the reversing of the capacitive phase for the equivalent circuit as the endpoint reached.

Appendix B BBR format Tool

The SignalHound BBR file format is relatively straightforward. A utility for Windows and Linux was written to transform these to CSV files for Excel so as to display the results of the plasma measurements.

The general flow for the processing is as per the diagram below.

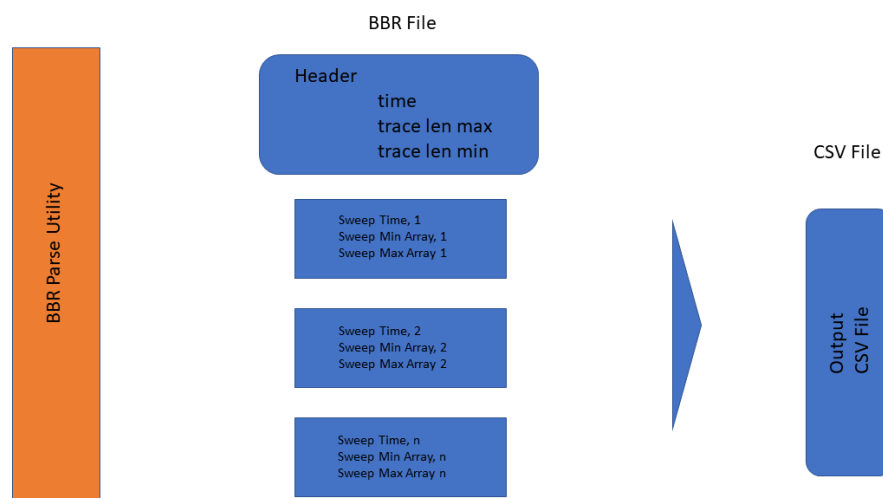


Figure 144 BBR Processing Flow Diagram

1. (Header 1 time only)
2. 64 bit integer, (standard ms from epoch time, when the sweep was captured)
3. header::trace_len 32-bit floats representing the “min” trace
4. header::trace_len 32-bit floats representing the “max” trace

(Repeat lines 2 to 3 until eof)

```
// *Begin*
// Header x1
// Sweep Time (sweep 1)
// Sweep Min Array (sweep 1)
// Sweep Max Array (sweep 1)
// Sweep Time (sweep 2)
// Sweep Min Array (sweep 2)
```

```
// Sweep Max Array (sweep 2)
// ... ( more sweeps )
// *End*
```

The Header comprises:-

```
typedef struct BBRHeader {
    unsigned short signature;
    unsigned short version;

    int sweep_count; // Number of sweeps in file

    unsigned short title[128];
    // Sweep configuration set by user during the time of sweep capture
    double center_freq;
    double span;
    double rbw;
    double vbw;
    double ref_level;
    double div;
    int atten;
    int gain;
    int detector;

    // Size of sweep
    int trace_len;
    // Start frequency of first bin in sweep
    double trace_start_freq;
    // Frequency difference between bins in sweep
    double bin_size;
}
```

The code block is as follows

```
// bbr_parse.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"

// This is an example of reading in a .bbr file generated by a sweep recording in Spike
// The general file structure of a .bbr file is below.

// *Begin*
// Header x1
// Sweep Time (sweep 1)
// Sweep Min Array (sweep 1)
// Sweep Max Array (sweep 1)
// Sweep Time (sweep 2)
// Sweep Min Array (sweep 2)
// Sweep Max Array (sweep 2)
// ... ( more sweeps )
// *End*
```

```

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
// #include <linux/types.h>

// #define __int64 int64_t

#define SM_BUF 1024
#define LM_BUF 8192

typedef struct BBRHeader {
    unsigned short signature;
    unsigned short version;

    int sweep_count; // Number of sweeps in file

    unsigned short title[128];
    // Sweep configuration set by user during the time of sweep capture
    double center_freq;
    double span;
    double rbw;
    double vbw;
    double ref_level;
    double div;
    int atten;
    int gain;
    int detector;

    // Size of sweep
    int trace_len;
    // Start frequency of first bin in sweep
    double trace_start_freq;
    // Frequency difference between bins in sweep
    double bin_size;
} BBRHeader;

//-----

void usage(void) {
    printf("Usage .. \n");
    printf("$.bbr_parse bbr_data_file output_csv_file\n\n");
}

//-----

void log_header(BBRHeader *h, FILE *f) {
    char op[SM_BUF];
    int res;

    res = snprintf(op, SM_BUF, " Signature      -> %u\n", h->signature);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Version      -> %u\n", h->version);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Sweep Count    -> %d\n", h->sweep_count);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Centre Frequency  -> %lf\n", h->center_freq);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Span Frequency   -> %lf\n", h->span);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Resolution BW    -> %lf\n", h->rbw);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Video BW        -> %lf\n", h->vbw);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Ref level       -> %lf\n", h->ref_level);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Div            -> %lf\n", h->div);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Atten          -> %d\n", h->atten);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Gain           -> %d\n", h->gain);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Detector        -> %d\n", h->detector);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Trace length    -> %d\n", h->trace_len);
    fwrite(op, 1, res, f);

```

```

        res = snprintf(op, SM_BUF, " Trace start freq   -> %lf\n", h->trace_start_freq);
        fwrite(op, 1, res, f);
        res = snprintf(op, SM_BUF, " Trace bin freq    -> %lf\n", h->bin_size);
        fwrite(op, 1, res, f);
    }

//-----

void dump_header(BBRHeader *h) {
    printf(" Signature      -> %u\n", h->signature);
    printf(" Version        -> %u\n", h->version);
    printf(" Sweep Count     -> %d\n", h->sweep_count);
    printf(" Centre Frequency -> %lf\n", h->center_freq);
    printf(" Span Frequency   -> %lf\n", h->span);
    printf(" Resolution BW    -> %lf\n", h->rbw);
    printf(" Video BW        -> %lf\n", h->vbw);
    printf(" Ref level       -> %lf\n", h->ref_level);
    printf(" Div            -> %lf\n", h->div);
    printf(" Atten          -> %d\n", h->atten);
    printf(" Gain           -> %d\n", h->gain);
    printf(" Detector        -> %d\n", h->detector);
    printf(" Trace length    -> %d\n", h->trace_len);
    printf(" Trace start freq -> %lf\n", h->trace_start_freq);
    printf(" Trace bin freq   -> %lf\n", h->bin_size);
}

//-----

void parse_bbr_file(int argc, char * argv[]) {
    FILE *fin = NULL;
    FILE *fout = NULL;
    float *min = NULL;
    float *max = NULL;
    BBRHeader header;
    __int64 time;
    int i, j;
    int res;
    char op[SM_BUF];

    printf("Berkeley BioScientific\n");

    if (argc < 3) {
        usage();
        exit(-1);
    }

    // f = fopen(argv[1], "rb");
    res = fopen_s(&fin, argv[1], "rb");
    if (!fin) {
        perror("Could not find file");
        exit(-2); // Error
    }

    res = fopen_s(&fout, argv[2], "wb");
    if (!fout) {
        perror("Could not open output file");
        exit(-3); // Error
    }

    // Read in header
    fread(&header, sizeof(BBRHeader), 1, fin);

    // Allocate arrays to hold sweep data
    min = (float*)malloc(sizeof(float) * header.trace_len);
    max = (float*)malloc(sizeof(float) * header.trace_len);

    dump_header(&header);
    log_header(&header, fout);

```

```

// Read in sweeps
for (i = 0; i < header.sweep_count; i++) {
    // Get sweep time stamp, epoch time in milliseconds
    fread(&time, sizeof(__int64), 1, fin);
    // Read the min and max sweep arrays
    // For average detectors, min array == max array
    fread(min, sizeof(float) * header.trace_len, 1, fin);
    fread(max, sizeof(float) * header.trace_len, 1, fin);

    // Do something with data
    // For most applications, the max sweep array is the useful array.
    // Use BBRHeader.trace_start_freq and BBRHeader.bin_size to get the frequency of a given
    // index into the sweep array.

    printf("%lld", time);

    res = snprintf(op, SM_BUF, "%lld", time);
    fwrite(op, 1, res, fout);

    for (j = 0; j < header.trace_len; j++) {
        printf(", %f", max[j]);

        res = snprintf(op, SM_BUF, ", %f", max[j]);
        fwrite(op, 1, res, fout);

    }
    printf("\n");

    res = snprintf(op, SM_BUF, "\n");
    fwrite(op, 1, res, fout);

}

// Done with file
if (fin) {
    fclose(fin);
    fin = NULL;
}
if (fout) {
    fclose(fout);
    fout = NULL;
}
if (min) {
    free(min);
    min = NULL;
}
if (max) {
    free(max);
    max = NULL;
}
}

//-----

int main(int argc, char *argv[]) {
    parse_bbr_file(argc, argv);
    printf("Done\n");
    return 0;
}

```

Appendix C TekTronix RSA306 format Tool

The Tektronix text format produces tagged columns of records. A utility for Windows and Linux was written to transform these to row-oriented CSV files for Excel so as to display the results of the plasma measurements.

```
// tek_parse.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

// This is an example of reading in a .tek file generated by a sweep recording
// The general file structure of a .tek file is below.

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
// #include <linux/types.h>

// #define __int64 int64_t

#define SM_BUF 1024
#define LM_BUF 8192

typedef struct TEKHeader {
    char spectrogram_time[SM_BUF];
    double frequency;
    double ref_level;
} TEKHeader;

typedef struct ResultsHeader {
    float RBW;
    double p1;
    double p2;
} ResultsHeader;

typedef struct SpectrogramTrace {
    int numberofpoints;
    double start_frequency;
    double end_frequency;
    double bucket_frequency;
    double timestamp;
} SpectrogramTrace;

//-----

void usage(void) {
    printf("Usage .. \n");
    printf("$ ./tek_parse tek_data_file output_csv_file\n\n");
}

//-----

void log_theader(TEKHeader *h, FILE *f) {
    char op[SM_BUF];
    int res;

    res = sprintf(op, SM_BUF, " Date/Time          -> %s", h->spectrogram_time);
    fwrite(op, 1, res, f);
    res = sprintf(op, SM_BUF, " Frequency          -> %lf\n", h->frequency);
    fwrite(op, 1, res, f);
    res = sprintf(op, SM_BUF, " Ref Level          -> %lf\n", h->ref_level);
    fwrite(op, 1, res, f);
}
```

```

}

//-----

void log_record(SpectrogramTrace *s, double *d, FILE *f) {
    char op[LM_BUF];

    int res;

    int i;

    res = snprintf(op, LM_BUF, "%lf", s->timestamp);
    fwrite(op, 1, res, f);

    if (s->numberofpoints > 1) {
        for (i = 0; i < s->numberofpoints - 1; i++) {
            res = res = snprintf(op, LM_BUF, "%lf", d[i]);
            fwrite(op, 1, res, f);
        }
        res = res = snprintf(op, LM_BUF, "%lf\n", d[s->numberofpoints - 1]);
        fwrite(op, 1, res, f);
    }

}

//-----

void dump_theader TEKHeader *h) {

    printf(" Date/Time      -> %s", h->spectrogram_time);
    printf(" Frequency      -> %lf\n", h->frequency);
    printf(" Ref Level        -> %lf\n", h->ref_level);
}

//-----

void dump_sheader(SpectrogramTrace *s) {

    printf(" Number of points  -> %d\n", s->numberofpoints);
    printf(" Start Frequency   -> %lf\n", s->start_frequency);
    printf(" End Frequency     -> %lf\n", s->end_frequency);
    printf(" Frequency bucket  -> %lf\n", s->bucket_frequency);

}

//-----

void log_sheader(SpectrogramTrace *s, FILE *f) {
    char op[SM_BUF];
    int res;
    int i = 0;
    double step;

    res = snprintf(op, SM_BUF, " Date/Time      -> %d\n", s->numberofpoints);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Start Frequency   -> %lf\n", s->start_frequency);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " End Frequency     -> %lf\n", s->end_frequency);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " Frequency steps  -> %lf\n", s->bucket_frequency);
    fwrite(op, 1, res, f);
    res = snprintf(op, SM_BUF, " ----- \n");
    fwrite(op, 1, res, f);

    res = snprintf(op, LM_BUF, "Timestamp,");
    fwrite(op, 1, res, f);

    if (s->numberofpoints > 1) {
        for (i = 0; i < s->numberofpoints - 1; i++) {
            step = s->start_frequency + (i * (s->bucket_frequency));
            res = res = snprintf(op, LM_BUF, "%lf", step);
            fwrite(op, 1, res, f);
        }
        step = s->start_frequency + (i * (s->bucket_frequency));
    }
}

```

```

        res = res = snprintf(op, LM_BUF, "%lf\n", step);
        fwrite(op, 1, res, f);
    }

//-----

void parse_tek_file(int argc, char * argv[]) {
    FILE *fin = NULL;
    FILE *fout = NULL;

    TEKHeader theader;
    // ResultsHeader rheader;
    SpectrogramTrace strace;

    int i;
    int lines = 0;
    int res;
    char op[SM_BUF];
    char ip[SM_BUF];
    int spectrogramline = 0;

    // hold record of results
    double sample[LM_BUF * 4];

    // pickup parameters of record
    int temp_numberofpoints;
    double temp_start_frequency;
    double temp_end_frequency;
    double temp_timestamp;
    double temp_bucket_frequency;

    // begin
    printf("Berkeley BioScientific\n");

    if (argc < 3) {
        usage();
        exit(-1);
    }

    res = fopen_s(&fin, argv[1], "rb");
    if (!fin) {
        perror("Could not find file");
        exit(-2); // Error
    }

    res = fopen_s(&fout, argv[2], "wb");
    if (!fout) {
        perror("Could not open output file");
        exit(-3); // Error
    }

    // Search for and read in Global header
    // This is going to be a very crude re-read search

    memset(theader.spectrogram_time, 0, SM_BUF);
    while (true) {
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 12);
            if (!strcmp("Spectrogram", op)) { // check for Spectrogram
                strcpy_s(theader.spectrogram_time, SM_BUF, &ip[12]);
                printf("%d found Timestamp -> %s", lines, theader.spectrogram_time);
                goto find_freq;
            }
        }
        else {
            fprintf(stderr, "Could not find timestamp before EOF\n");
        }
    }
}

```



```

        goto done;
    }
}

find_freq:

    // Find the Frequency, should follow

    theader.frequency = 0.0;
    while (true) {
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 10);
            if (!strcmp("Frequency,", op)) { // check for Spectrogram
                res = sscanf_s(&ip[10], "%lf", &theader.frequency);
                printf("%d found Frequency -> %lf\n", lines, theader.frequency);
                goto find_ref;
            }
        }
        else {
            fprintf(stderr, "Could not find Frequency before EOF\n");
            goto done;
        }
    }

find_ref:

    // Find the Reference Level, should follow

    theader.ref_level = 0.0;
    while (true) {
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 16);
            if (!strcmp("Reference Level,", op)) { // check for Spectrogram
                res = sscanf_s(&ip[16], "%lf", &theader.ref_level);
                printf("%d found Reference Level -> %lf\n", lines, theader.ref_level);
                goto find_traces;
            }
        }
        else {
            fprintf(stderr, "Could not find Reference Level before EOF\n");
            goto done;
        }
    }

find_traces:

    dump_theader(&theader);
    log_theader(&theader, fout);

    while (true) { // keep reading remainder of file
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 15);
            if (!strcmp("SpectrogramLine", op)) { // check for Spectrogram

                printf("%d found Spectrogram Line -> %d\n", lines, spectrogramline);
                spectrogramline++;

                // Now get the samples

                // Number of points first
                memset(ip, 0, SM_BUF); // clear buffer
                memset(op, 0, SM_BUF);

```

```

        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 13);
            if (!strcmp("NumberPoints,", op)) { // check for number of samples
                res = sscanf_s(&ip[13], "%d", &temp_numberofpoints);
                printf("%d Processing Samples -> %d\n", lines,
temp_numberofpoints);
            }
        }
        // sanity check
        if ((LM_BUF * 4) < temp_numberofpoints) {
            fprintf(stderr, "Must allocate more samples space
%d:%d\n", LM_BUF * 4, temp_numberofpoints);
            exit(1);
        }

        // Start frequency
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 7);
            if (!strcmp("XStart,", op)) { // check for start frequency following
                res = sscanf_s(&ip[7], "%lf", &temp_start_frequency);
                printf("%d Processing Samples starting at -> %lf\n",
lines, temp_start_frequency);
            }
        }
        // Stop frequency
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 6);
            if (!strcmp("XStop,", op)) { // check for start frequency following
                res = sscanf_s(&ip[6], "%lf", &temp_end_frequency);
                printf("%d Processing Samples ending at -> %lf\n",
lines, temp_end_frequency);
            }
        }

        // Timestamp
        memset(ip, 0, SM_BUF); // clear buffer
        memset(op, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            memcpy(op, ip, 10);
            if (!strcmp("Timestamp,", op)) { // check for start frequency
                res = sscanf_s(&ip[10], "%lf", &temp_timestamp);
                printf("%d Processing Samples timestamp at -> %lf\n",
lines, temp_timestamp);
            }
        }

        temp_bucket_frequency = (temp_end_frequency - temp_start_frequency) /
(temp_numberofpoints - 1);
        printf("%d The frequency jump of each bucket -> %lf\n", lines,
temp_bucket_frequency);

        strace.timestamp = temp_timestamp; // This is independent

        if (1 == spectrogramline) { // first spectrogram line
            printf("%d Assigning record parameters for Spectrogram %d\n",
lines, spectrogramline);

            strace.numberofpoints    = temp_numberofpoints;
            strace.start_frequency   = temp_start_frequency;
            strace.end_frequency     = temp_end_frequency;
            strace.bucket_frequency = temp_bucket_frequency;
            dump_sheader(&strace);
            log_sheader(&strace, fout);
        }
        else { // check if they have changed.. notify user

```

```

        printf("%d Subsequent records, check record parameters\n",
lines);
        if (temp_numberofpoints != strace.numberofpoints) {
            fprintf(stderr, "Detected a change in Number of Points
%d:%d\n", temp_numberofpoints, strace.numberofpoints);
            goto done;
        }
        if (temp_start_frequency != strace.start_frequency) {
            fprintf(stderr, "Detected a change in Start frequency
%lf:%lf\n", temp_start_frequency, strace.start_frequency);
            goto done;
        }
        if (temp_end_frequency != strace.end_frequency) {
            fprintf(stderr, "Detected a change in End frequency
%lf:%lf\n", temp_end_frequency, strace.end_frequency);
            goto done;
        }
        if (temp_bucket_frequency != strace.bucket_frequency) {
            fprintf(stderr, "Detected a change in Bucket frequency
%lf:%lf\n", temp_bucket_frequency, strace.bucket_frequency);
            goto done;
        }
    }

    // so now acquire them !
    for (i = 0; i < strace.numberofpoints; i++) {
        memset(ip, 0, SM_BUF);
        if (fgets(ip, sizeof(ip), fin) != NULL) { // get a line or partial or exit
            lines++;
            res = sscanf_s(ip, "%lf", &sample[i]);
            if (!res) {
                fprintf(stderr, "%d Unexpected error reading a
sample %s\n", lines, ip);
            }
        }
        else {
            fprintf(stderr, "%d Unexpected error or EOF while
reading samples\n", lines);
            goto done;
        }
    }

    printf("Trace %d sample 0 -> %lf\n", spectrogramline, sample[0]);
    printf("Trace %d sample %d -> %lf\n", spectrogramline,
strace.numberofpoints-1, sample[strace.numberofpoints-1]);

    // put to file
    log_record(&strace, sample, fout);

    }
    else {
        if (spectrogramline) {
            printf("Processed %d Spectrogram Records\n", spectrogramline);
        }
        else {
            fprintf(stderr, "Could not find Spectrogram Line before EOF\n");
        }
        goto done;
    }
}

done:

// Done with file
if (fin) {
    fclose(fin);
    fin = NULL;
}

```

```

    }
    if (fout) {
        fclose(fout);
        fout = NULL;
    }

    printf("Processed %d input lines\n", lines);
}

//-----

int main(int argc, char *argv[]) {
    parse_tek_file(argc, argv);
    printf("Done\n");
    return 0;
}

```

Appendix D Ocean Optics Acquisition Subsystem for Windows®

The Ocean Optics subsystem has an API for extracting the real time records from the spectrometer. We have included the basic module for acquiring from the HR4000 and the USB2K devices under Windows®.

```
#include <windows.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <io.h>
#include <tchar.h>
#include <stdio.h>
#include <time.h>
#include <strsafe.h>
#include <Winerror.h>
#include "mosquitto.h"
#include <process.h>

#include "api/SeaBreezeWrapper.h"

#define MAX_LABEL_SIZE 15 /* largest length of any spectrometer type or serial number */
#define MAX_DEVICES 16 /* max number of connected spectrometers */

#define VERSION "V1.1.1"

#define SM_BUF 1024
#define MQ_BUF 128*1024 /* 4K samples at 32 bit floats ~ 16K .. float as ascii 2 decimal places */
/* returns doubles, but we may need to truncate to float */
*/
#define OCEAN_OPTICS_MAX 4096 /* HRC4000 only produces max ~3648 pixels .. allow for 4K */

#define RUNNING 2
#define STOPPED 1

#define LINK_DOWN 0
#define LINK_UP 1

#define KEEPALIVE 60
#define MQTT_PORT 1883

#define F_EXIST 00
#define F_WRITE_PERMISSION 01
#define F_READ_PERMISSION 04
#define F_READ_WRITE_PERMISSION 6

//-----
// Macros

#if 0
#define VPRINT(A)
#define VDPRINT(A)
#else
#define VPRINT(fmt, ...) do { if (flag_v) { printf (fmt, ##__VA_ARGS__); } } while (0)
#define VDPRINT(fmt, ...) do { if (flag_v && flag_D) { printf (fmt, ##__VA_ARGS__); } } while (0)
#endif

//-----

// Infile stuff
char infile[SM_BUF] = "oes0.ini";
```

```

char *arg_infile = infile;

// Logfile stuff
char logfile[SM_BUF] = "OES_logfile";
FILE *lf = NULL;
int logging = 0;
char logday[SM_BUF] = "";
int file_flush = 0;

// Mosquitto
char sbroker[SM_BUF] = "localhost";
char clientid[SM_BUF];
int mqport;
char topicname[SM_BUF] = "ICE121/OE0";
char sensorname[SM_BUF] = "OE0";
int persist = 0;
int keepalive;
int link_status = LINK_DOWN;

// Thread and Mutex
HANDLE data_lock;
HANDLE mq_worker;
DWORD ThreadID;

// Licence
char licence[SM_BUF];

volatile int keep_going = 1;
volatile int mode = STOPPED; // lock for DO

// usage flags
int flag_t = 0; /* Test mode */
int flag_v = 0; /* verbose */
int flag_D = 0; /* debug */
int flag_i = 0; /* infile .. mandatory */

// struct spectrometer {
    char devname[SM_BUF];
    char serialnumber[SM_BUF];
    unsigned long integrationtime;
    int boxcarwidth;
    int scanperavg;
// };

int thresholdenable = 0;
float thresholdvalue = 0.0;

int spectrometer_index = 0;

// typedef struct spectrometer spectrometer_t;

long long total_read_count = 0;

//-----
//-----

BOOL WINAPI consoleHandler(DWORD signal) {
    switch (signal) {
        case CTRL_C_EVENT:
            keep_going = false;
            printf("Ctrl-C handled\n"); // do cleanup
            return TRUE;
        default:
            printf("Event [%d] handled\n", signal);
            return FALSE;
    }
}

//-----
//-----

```

```

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n", (void*)&mosq, obj, result);
}

//-----

void message_callback(struct mosquitto *mosq, void *obj,
    const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n", (void*)&mosq,
        (void*)&obj, (void*)&message);
}

//-----
//-----

void mosquitto_error(int er) {
    switch (er) {
        case MOSQ_ERR_CONN_PENDING:
            VPRINT("Mosquitto erro Pending\n");
            break;
        case MOSQ_ERR_SUCCESS:
            VPRINT("Mosquitto erro Success\n");
            break;
        case MOSQ_ERR_NOMEM:
            VPRINT("Mosquitto erro NoMem\n");
            break;
        case MOSQ_ERR_PROTOCOL:
            VPRINT("Mosquitto erro ERR Protocol\n");
            break;
        case MOSQ_ERR_INVALID:
            VPRINT("Mosquitto erro Invalid\n");
            break;
        case MOSQ_ERR_NO_CONN:
            VPRINT("Mosquitto erro No CONN\n");
            break;
        case MOSQ_ERR_CONN_REFUSED:
            VPRINT("Mosquitto erro CONN refused\n");
            break;
        case MOSQ_ERR_NOT_FOUND:
            VPRINT("Mosquitto erro Not found\n");
            break;
        case MOSQ_ERR_CONN_LOST:
            VPRINT("Mosquitto erro CONN lost\n");
            break;
        case MOSQ_ERR_TLS:
            VPRINT("Mosquitto erro TLS\n");
            break;
        case MOSQ_ERR_PAYLOAD_SIZE:
            VPRINT("Mosquitto erro payload size\n");
            break;
        case MOSQ_ERR_NOT_SUPPORTED:
            VPRINT("Mosquitto erro not supported\n");
            break;
        case MOSQ_ERR_AUTH:
        case MOSQ_ERR_ACL_DENIED:
            VPRINT("Mosquitto erro ACL denied\n");
            break;
        case MOSQ_ERR_UNKNOWN:
            VPRINT("Mosquitto erro unknown\n");
            break;
        case MOSQ_ERR_ERRNO:
            VPRINT("Mosquitto connect before loop start\n");
            break;
        case MOSQ_ERR_EAI:
            VPRINT("Mosquitto erro EAI, getinfoaddr\n");
            break;
        default:
            VPRINT("Mosquitto default - no error defined\n");
            break;
    }
}

//-----

```

```

char tohex(int chr) {
    chr &= 0x0f;
    switch (chr) {
        case 0x00: return '0'; case 0x01: return '1';
        case 0x02: return '2'; case 0x03: return '3';
        case 0x04: return '4'; case 0x05: return '5';
        case 0x06: return '6'; case 0x07: return '7';
        case 0x08: return '8'; case 0x09: return '9';
        case 0x0a: return 'a'; case 0x0b: return 'b';
        case 0x0c: return 'c'; case 0x0d: return 'd';
        case 0x0e: return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}

//-----
void ErrorExit(LPTSTR lpszFunction) {
    // Retrieve the system error message for the last-error code

    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf,
        0, NULL);

    // Display the error message and exit the process

    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR)lpMsgBuf) + lstrlen((LPCTSTR)lpszFunction) + 40) * sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(TCHAR),
        TEXT("%s failed with error %d: %s"),
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
    // ExitProcess(dw);
}

//-----

unsigned long long current_timestamp(int flag) {
    ULARGE_INTEGER theTime;
    unsigned __int64 fileTime64Bit;
    FILETIME ft;

    GetSystemTimePreciseAsFileTime(&ft); // returns highest precision time
    theTime.LowPart = ft.dwLowDateTime; // number of 100nS intervals since
    theTime.HighPart = ft.dwHighDateTime; // Jan 1 1601(UTC)
    fileTime64Bit = theTime.QuadPart;

    return (fileTime64Bit);
}

//-----

int wr3_day(char * b, int len) {
    time_t t;
    struct tm tm;
    int wday;
    char *sday[7] = { "Sun", "Mon", "Tue", "Wed", "Thurs", "Fri", "Sat" };

    t = time(NULL);
    localtime_s(&tm, &t);

```



```

        wday = tm.tm_wday;
        snprintf(b, len, "%s", sday[wday]);
        return 0;
    }

//-----

int wr3_date(char *tim_buf, int len, int dec_places) {

#define D_FORMAT "%A"
#define TDZ_FORMAT "%z"
#define YMDHMS_FORMAT "%Y-%m-%dT%H:%M:%S."
#define BUF 64

    __time64_t ltm;
    struct tm tm;
    errno_t err;
    size_t sl; // chars returned
    char tdz[BUF]; // timezone
    SYSTEMTIME st;
    int res;

    _tzset();
    _time64(&ltm);
    err = _localtime64_s(&tm, &ltm);
    GetSystemTime(&st);

    // Get time zone difference as "xxxx"
    // such that can expand to "xx:xx"
    sl = strftime(tdz, sizeof(tdz), TDZ_FORMAT, &tm);
    if (0 == sl)
    {
        fprintf(stderr, "strftime return 0\n");
        return -2;
    }
    // simple assignation of the 5 bytes for format required
    tdz[6] = 0; // zero terminate
    tdz[5] = tdz[4];
    tdz[4] = tdz[3];
    tdz[3] = ':';
    // printf("Timezone is %s\n", tdz);

    res = snprintf(tim_buf, len,
        //yyyy-mm-ddThh:mm:ss.fff+01:00
        "%4d-%2d-%2dT%2d:%2d:%2d.%3d%s",
        st.wYear,
        st.wMonth,
        st.wDay,
        st.wHour,
        st.wMinute,
        st.wSecond,
        st.wMilliseconds,
        tdz);

    // printf("OUTBUF is -> '%s'\n", tim_buf);

    return 0;
}

//-----

int build_shaper_mq_header(char *buf,
    int buf_len,
    char *sensor_name,
    char *data_type,
    char *data_structure,
    int elements,
    char *separator,
    int msg_len) {

    int res;
    char tim_buf[SM_BUF];

```

```

    res = wr3_date(tim_buf, SM_BUF, 4/* decimal places*);
    if (res) {
        fprintf(stderr, "Error in datetime function\n");
    }

    res = snprintf(buf, buf_len, "%s;%s;%s;%s;%d;%s;%d",
        sensor_name,
        tim_buf,
        data_type,
        data_structure,
        elements,
        separator,
        msg_len);

    if (buf_len < res) {
        fprintf(stderr, "Could not pack header, buf too small %d:%d\n", buf_len, res);
        return -1;
    }
    return res;
}

//-----

// Calc checksum to insert in 'three digit [000 to 255] checksum and ':'
int calc_tail(char * mq_buf, int mlen) {
    int i;
    int sum = 0;

    for (i = 0; i < mlen - 4; i++) { // exclude already add 'digits' and ':'
        sum += mq_buf[i];
    }
    sum &= 0xff;
    snprintf(&mq_buf[mlen - 4], 4, "%03d", sum); // left justified 3 digits decimal

    return 0;
}

//-----

void usage() {
    puts("Usage: usb2k4k -c infile\n"
        "Examples:\n"
        "  $ usb2k4k -c oes0.in\n");
    exit(1);
}

//-----

char * get_error_string(int a) {
    if (a) {
        printf("error called %d\n", a);
        return ("oops");
    }
    return (NULL);
}

//-----

int load_ini_parms(TCHAR * ini_file) {
    DWORD res;
    TCHAR oesunit[SM_BUF];
    TCHAR _clientid[SM_BUF]; // temp space
    TCHAR sAbsolutePathBuffer[2048] = _T(""); // getprofile requires full pathname !!

    GetFullPathName(ini_file, sizeof(sAbsolutePathBuffer) / sizeof(TCHAR), sAbsolutePathBuffer, NULL);

    // General stuff
    VDPRINT(("Read --> %s\n", "General:Logfile"));
    res = GetPrivateProfileString(_T("General"), _T("Logfile"), _T("OES"), oesunit, SM_BUF,
sAbsolutePathBuffer);
    _stprintf_s(logfile, MQ_BUF, _T("%s"), oesunit);

    VDPRINT(("Read --> %s\n", "General:Logging"));

```

```

        res = GetPrivateProfileString(_T("General"), _T("Logging"), _T("0"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        logging = !!(atoi(oesunit));

        VDPRINT(("Read --> %s\n", "General:Licence"));
        res = GetPrivateProfileString(_T("General"), _T("Licence"), _T("Not Licensed"), licence, SM_BUF,
sAbsolutePathBuffer);

        VDPRINT(("Read --> %s\n", "General:Flush"));
        res = GetPrivateProfileString(_T("General"), _T("Flush"), _T("0"), oesunit, SM_BUF, sAbsolutePathBuffer);
        file_flush = !!(atoi(oesunit));

        // Spectrometer stuff
        VDPRINT(("Read --> %s\n", "Spectrometer:device"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("device"), _T("NoDevice"), devname, SM_BUF,
sAbsolutePathBuffer);

        VDPRINT(("Read --> %s\n", "Spectrometer:sensorname"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("sensorname"), _T("OES0"), sensorname, SM_BUF,
sAbsolutePathBuffer);

        VDPRINT(("Read --> %s\n", "Spectrometer:integrationtime"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("integrationtime"), _T("87000"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        integrationtime = atoi(oesunit);

        VDPRINT(("Read --> %s\n", "Spectrometer:boxcarwidth"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("boxcarwidth"), _T("0"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        boxcarwidth = atoi(oesunit);

        VDPRINT(("Read --> %s\n", "Spectrometer:scanperavg"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("scanperavg"), _T("1"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        scanperavg = atoi(oesunit);

        VDPRINT(("Read --> %s\n", "Spectrometer:ThresholdEnable"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("Thresholdenable"), _T("0"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        thresholdenable = !!(atoi(oesunit));

        VDPRINT(("Read --> %s\n", "Spectrometer:ThresholdValue"));
        res = GetPrivateProfileString(_T("Spectrometer"), _T("ThresholdValue"), _T("0"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        res = sscanf_s(oesunit, "%f", &thresholdvalue);
        if (1 != res) {
            thresholdvalue = 2000.0;
            fprintf(stderr, "Could not read Threshold Value '%s', defaulting to %.2f\n", oesunit,
thresholdvalue);
        }

        // Mosquitto Stuff
        VDPRINT(("Read --> %s\n", "Mosquitto:topicname"));
        res = GetPrivateProfileString(_T("Mosquitto"), _T("topicname"), _T("OES"), topicname, SM_BUF,
sAbsolutePathBuffer);

        // set up a reasonable random client id (can pass NULL to procedure!)
        memset(_clientid, 0, 24); // generate a temporary one using PID
        snprintf(_clientid, 23, "oes_%d", GetProcessId(NULL));

        VDPRINT(("Read --> %s\n", "Mosquitto:clientid"));
        // only 24 characters permitted for 'client_id' // leave 1 for 'NULL'
        res = GetPrivateProfileString(_T("Mosquitto"), _T("clientid"), _clientid, clientid, 23, sAbsolutePathBuffer);

        VDPRINT(("Read --> %s\n", "Mosquitto:broker"));
        res = GetPrivateProfileString(_T("Mosquitto"), _T("broker"), _T("localhost"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        snprintf(sbroker, SM_BUF, oesunit);

        VDPRINT(("Read --> %s\n", "Mosquitto:mqport"));
        res = GetPrivateProfileString(_T("Mosquitto"), _T("mqport"), _T("MQTT_PORT"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        mqport = _ttoi(oesunit);

```

```

        VDPRINT(("Read --> %s\n", "Mosquitto:keepalive"));
        res = GetPrivateProfileString(_T("Mosquitto"), _T("keepalive"), _T("KEEPALIVE"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        keepalive = atoi(oesunit);

        VDPRINT(("Read --> %s\n", "Mosquitto:persist"));
        res = GetPrivateProfileString(_T("Mosquitto"), _T("persist"), _T("0"), oesunit, SM_BUF,
sAbsolutePathBuffer);
        keepalive = !!atoi(oesunit);

        return 0;
}

//-----

void flags_init(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (!strcmp(argv[i], "-t")) { flag_t = 1; continue; }
        if (!strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (!strcmp(argv[i], "-c") && (i + 1 < argc)) {
            flag_i = 1; arg_inifile = argv[++i]; continue;
        }

        fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
        usage();
    }

    if (flag_i && arg_inifile) {
        if (strlen(arg_inifile) < sizeof(inifile)) {
            snprintf(inifile, SM_BUF, "%s", arg_inifile);
            if (_access_s(arg_inifile, F_READ_PERMISSION) != -1) {
                // file exists
                VPRINT("Found ini file\n");
            }
            else {
                // file doesn't exist
                fprintf(stderr, "Could find ini file\n");
            }
        }
    }

    if (!flag_i) {
        fprintf(stderr, "You must specify an INI file\n");
        usage(); // does not return
    }

    load_ini_parms(inifile); // read some ini parms

    printf("Inifile is      --> %s\n", inifile);
    printf("Licence is         --> %s\n", licence);
    printf("Broker is          --> %s:%d\n", sbroker, mqport);
    printf("ClientID is        --> %s\n", clientid);
    printf("Topic is           --> %s\n", topicname);
    printf("Sensor is          --> %s\n", sensorname);
    printf("Device is          --> %s\n", devname);
    printf("Debug is           --> %d\n", flag_D);
    printf("Verbose is         --> %d\n", flag_v);
    printf("Test Mode is       --> %d\n", flag_t);

    printf("Logging is         --> %d\n", logging);
    printf("Logfile is         --> %s\n", logfile);
    printf("Logfile Flush is   --> %d\n", file_flush);
    printf("Threshold enabled   --> %s\n", thresholdenable ? "Yes" : "No");
    printf("Threshold value     --> %f\n", thresholdvalue);

    fflush(NULL);
}

//-----

```

```

//-----

// Only does a 'loop' when semaphore is set from main
// which reads the data from the OES
//
static void * post_thread(void *arg) {

    char mq_buf[MQ_BUF];
    char header_buf[SM_BUF];
    char data_buf[MQ_BUF];
    char *user_ptr = NULL;
    char svalue[32];

    int res;
    int dl; // data length
    int ml; // message length
    int i;

    int error;
    int spec_length;
    int pixels_read;
    double spectrum[OCEAN_OPTICS_MAX];
    double wavelengths[OCEAN_OPTICS_MAX];

    double peak_spectrum_value;
    int peak_spectrum_index;

    struct mosquitto *mosq = NULL;

    VPRINT(">>Entered Worker thread with argp -> %p\n", arg);

    mosquitto_lib_init();
    mosq = mosquitto_new(clientid, 1, user_ptr);
    if (mosq) {
        mosquitto_connect_callback_set(mosq, connect_callback);
        mosquitto_message_callback_set(mosq, message_callback);
        res = mosquitto_connect(mosq, sbroker, mqport, keepalive);
        if (res) {
            mosquitto_error(res);
            printf("%s Link Down %d\n", sensorname, res);
            link_status = LINK_DOWN;
        }
        else {
            printf("%s Link UP\n", sensorname);
            link_status = LINK_UP;
        }
    }
    else {
        fprintf(stderr, "Could not start Mosquitto\n");
        exit(1);
    }
    // start network loop thread
    mosquitto_loop_start(mosq);

    spec_length = seabreeze_get_formatted_spectrum_length(spectrometer_index, &error);
    pixels_read = seabreeze_get_wavelengths(spectrometer_index, &error, wavelengths, spec_length);
    // Tell the user the wavelengths of this spectrometer .. give a sample
    for (i = 0; i < 10; i++) {
        printf("Wavelength[%d] = %.3f\n", i, wavelengths[i]);
    }

    for (i = pixels_read-10; i < pixels_read; i++) {
        printf("Wavelength[%d] = %.3f\n", i, wavelengths[i]);
    }

    // now main loop .. just wait semaphore to run and acquire sample
    while (keep_going) {
        WaitForSingleObject(data_lock, INFINITE);        // down semaphore, data is in

        // potential race here on data, but

```

```

// serial data is 'relatively' slow

// we dont 'critical section' the data

// as this may 'hold' up serial processing
if (keep_going) { // check one
more time, remove that last one !

// printf("Getting formatted spectrum from matching spectrometer (index %02d).\n",
spectrometer_index);
spec_length = seabreeze_get_formatted_spectrum_length(spectrometer_index,
&error);
if (0 != error) {
printf(stderr, " [%02d] Error getting spectrum length: %s\n",
spectrometer_index, get_error_string(error));
}
else {
// double *spectrum = (double *)calloc((size_t)spec_length, sizeof(double));
pixels_read = seabreeze_get_formatted_spectrum(spectrometer_index,
&error, spectrum, spec_length);
// printf(" [%02d] Read %d pixels (error code '%s').\n", spectrometer_index,
pixels_read, get_error_string(error));
// free(spectrum);
}

data_buf[0] = 0;
for (i = 0; i < pixels_read - 1; i++) {
snprintf(svalue, 32, "%f", spectrum[i]);
strcat_s(data_buf, MQ_BUF, svalue);
}
snprintf(svalue, 32, "%f", spectrum[pixels_read-1]); // final one .. no ','
strcat_s(data_buf, MQ_BUF, svalue);
dl = strlen(data_buf, MQ_BUF);

#if 0
for (i = 0; i < 10 /*pixels_read*/; i++) {
printf("%d -> %.2f, ", i, (float)spectrum[i]);
// dl += snprintf(&data_buf[i], MQ_BUF, "%f", (float)spectrum[i]);
}
printf("\n");

for (i = 1000; i < 1000+10 /*pixels_read*/; i++) {
printf("%d -> %.2f, ", i, (float)spectrum[i]);
// dl += snprintf(&data_buf[i], MQ_BUF, "%f", (float)spectrum[i]);
}
printf("\n");

#endif

peak_spectrum_value = spectrum[0];
peak_spectrum_index = 0;

for (i = 0; i < pixels_read; i++) {
if (peak_spectrum_value < spectrum[i]) {
peak_spectrum_value = spectrum[i];
peak_spectrum_index = i;
}
}

// printf("Peak spectrum @ %d = %f\n", peak_spectrum_index, peak_spectrum_value);

if (thresholdenable) {
if (thresholdvalue > peak_spectrum_value) { // didnt meet the requirement
goto skip;
}
}

build_shaper_mq_header(header_buf,
SM_BUF,
sensormame,
"CSV",
"float",
28,

```

```

        /* "\", \"", */
        " ",
        ',
        dl);

ml = snprintf(mq_buf, MQ_BUF, "%s|s|%03d|", header_buf, data_buf, 0xff);
calc_tail(mq_buf, ml);

switch (link_status) {
case LINK_UP:
    res = mosquitto_publish(mosq,
        NULL,
        topicname,
        strlen(mq_buf),
        mq_buf,
        0,
        0);

    if (res) { // not good !
        link_status = LINK_DOWN;
        printf("%s -> link down \n", topicname);
    }
    break;

case LINK_DOWN:
    res = mosquitto_reconnect(mosq); // broker down, attempt reconnect
    if (!res) {
        link_status = LINK_UP;
        printf("%s -> link up \n", topicname);
    }
    break;

default:
    fprintf(stderr, "%s Internal error on link status %s\n", topicname, sbroker);
    break;

} // link_status ?

if (flag_D) {
    printf("%s:%s\n", sensorname, mq_buf);
}

} // keep_going one last time
skip:
    ; // if thresholding enabled and we didnt meet requirement
} // while keep_going

mosquitto_disconnect(mosq);
mosquitto_destroy(mosq);
VPRINT("%s Disconnected from broker %s\n", topicname, sbroker);

return NULL;
}

//-----

int my_main(int argc, char **argv) {

    char *desired_type = NULL; // what spectrometer model we want
    char *desired_serial_number = NULL; // what spectrometer SN we want
    char spectrometer_model[SM_BUF]; // What is the device connected
    int matching_devices[MAX_DEVICES]; // indices of matching devices
    int connected_device_count = 0; // how many spectrometers found
    int matching_device_count = 0; // how many matching devices found
    int error = 0; // SeaBreeze error result code

    int min_integration = 0; // minimum integration time
    float spectrometer_temperature = 0.0;

    int i;
    char label[MAX_LABEL_SIZE + 1];
    // char type[MAX_LABEL_SIZE + 1];

```

```

int  opterr = 0;
int  c = 0;
int  res;
int  exit_value = EXIT_SUCCESS;
long long unsigned start_time, stop_time, delta_time;
int  loop_read_count = 0;
int  potential_match = 1;
// int  index;
int  spec_length = 0;

//-----

flags_init(argc, argv);
desired_serial_number = devname;

start_time = current_timestamp(0); // hundreds of nanoseconds since jan 1 1601
printf("Time starting is %lld\n", start_time);

printf("Finding connected optical spectrometers...\n");
for (i = 0; i < SEABREEZE_MAX_DEVICES; i++)
    if (0 == seabreeze_open_spectrometer(i, &error)) {
        connected_device_count++;
    }
    else
        break;
printf("%d devices found.\n", connected_device_count);

////////////////////
// look for desired spectrometer
////////////////////

// iterate over connected devices, attempting to match on model & serial
printf("Looking for matching optical spectrometer(s)...\n");
for (i = 0; i < connected_device_count; i++) {

    #if 0

        // match on spectrometer type
        seabreeze_get_model(i, &error, type, sizeof(type));
        if (0 == error) {
            if (desired_type) {
                if (!strcmp(type, desired_type)) {
                    printf(" [%02d] matches desired optical spectrometer type '%s'\n",
i, type);
                }
                else {
                    printf(" [%02d] does not match desired optical spectrometer type
'%s' (%s)\n", i, desired_type, type);
                    potential_match = 0;
                }
            }
            else {
                printf(" [%02d] has optical spectrometer type '%s' (no match criteria
provided)\n", i, type);
            }
        }
        else {
            printf(" [%02d] Error getting type: %s\n", i, get_error_string(error));
            potential_match = 0;
        }

    #endif

        // match on serial_number
        seabreeze_get_serial_number(i, &error, label, sizeof(label));
        if (0 == error) {
            if (desired_serial_number) {
                if (!strcmp(label, desired_serial_number, sizeof(label))) {
                    printf(" [%02d] matches desired serial number '%s'\n", i, label);
                    seabreeze_get_model(i, &error, spectrometer_model, SM_BUF);
                    printf(" The model attached is '%s'\n", spectrometer_model);
                    min_integration =
seabreeze_get_min_integration_time_microsec(i, &error);
                    printf(" Minimum integration time is %d\n", min_integration);

```



```

spectrometer_temperature = seabreeze_read_tec_temperature(i,
&error);
spectrometer_temperature);
integrationtime);
integrationtime);

    }
    else {
        printf(" [%02d] does not match desired serial number '%s'
(%s)\n", i, desired_serial_number, label);
        potential_match = 0;
    }
    }
    else {
        printf(" [%02d] has serial number '%s' (no match criteria provided)\n", i,
label);
    }
    }
    else {
        printf(" [%02d] Error getting serial number: %s\n", i, get_error_string(error));
        potential_match = 0;
    }
    // aggregate results
    if (potential_match) {
        printf(" [%02d] matched all parameters.\n", i);
        matching_devices[matching_device_count++] = i;
    }
    else {
        printf(" [%02d] failed to match your parameters (closing).\n", i);
        seabreeze_close_spectrometer(i, &error);
        if (0 != error) {
            printf(" [%02d] Error closing optical spectrometer: %s\n", i,
get_error_string(error));
        }
    }
    printf("\n");
}

// give up if no matches
if (!matching_device_count) {
    printf("Sorry, no matching devices found.\n");
    return -1;
}

////////////////////////////////////
// Process matching devices
////////////////////////////////////

// display matching index(es)
printf("Found %d matching devices, at the following indices:\n", matching_device_count);
for (i = 0; i < matching_device_count; i++)
    printf(" %d", matching_devices[i]);
printf("\n");

// take first device
spectrometer_index = matching_devices[0];

#if 0

// take one acquisition from each matching device
for (i = 0; i < matching_device_count; i++) {
    index = matching_devices[i];
    spec_length = 0;

    printf("Getting formatted spectrum from matching optical spectrometer #%%d (index %02d).\n", i,
index);
    spec_length = seabreeze_get_formatted_spectrum_length(index, &error);
    if (0 != error) {

```

```

        printf(" [%02d] Error getting optical spectrum length: %s\n", index,
get_error_string(error));
    }
    else {
        double *spectrum = (double *)calloc((size_t)spec_length, sizeof(double));
        int pixels_read = seabreeze_get_formatted_spectrum(index, &error, spectrum,
spec_length);
        printf(" [%02d] Read %d pixels (error code '%s').\n", index, pixels_read,
get_error_string(error));
        free(spectrum);
    }
}

#endif

```

```

data_lock = CreateSemaphore(
    NULL,                                // default security attributes
    0,                                    // Initial Count
    1,                                    // maximum count
    NULL);                                // unnamed semaphore

if (NULL == data_lock) {
    fprintf(stderr, "Could not create semaphore\n");
    exit_value = EXIT_FAILURE;
    goto done;
}

mq_worker = CreateThread(
    NULL,                                // default security attributes
    0,                                    // default stack size
    (LPTHREAD_START_ROUTINE)post_thread,
    NULL,                                // No tread functions arguments
    0,                                    // default creation flags
    &ThreadID);                            // receive thread identifier

keep_going = TRUE;

if (!SetConsoleCtrlHandler(consoleHandler, TRUE)) {
    fprintf(stderr, "Could not set control handler\n");
    exit_value = EXIT_FAILURE;
    goto done;
}

while (keep_going) {
    Sleep(100); //wait 500 mS
    total_read_count++;
    loop_read_count++;

    if (loop_read_count >= 5) { // time to estimate ?
        loop_read_count = 0;

        res = ReleaseSemaphore(
            data_lock,                    // handle to semaphore
            1,                            // increase count by one
            NULL);                        // not interested in previous count

        if (!res) {
            fprintf(stderr, "Error bumping semaphore %d\n", res);
        }

        } // want to decimate the data_rate !
    } // end while keep-going

```

done:

```

keep_going = 0; // force for debug, if we are jumping here

// cleanup
for (i = 0; i < matching_device_count; i++) {
    int index = matching_devices[i];
    seabreeze_close_spectrometer(index, &error);
    if (0 != error) {
        printf(" [%02d] Error closing optical spectrometer: %s\n", index,
get_error_string(error));
    }
}

VPRINT("No of records read was : %lld\n", total_read_count);

stop_time = current_timestamp(0); // hundreds of nanoseconds since jan 1 1601

printf("Time start      :      is %llu (100nS)\n", start_time);
printf("Time now       :      is %llu (100nS)\n", stop_time);

delta_time = stop_time - start_time;
printf("Delta time      :      is %llu (mS)\n", delta_time / (10 * 1000));
printf("Total loop count :      is %lld\n", total_read_count);

res = ReleaseSemaphore( // send one to free it !
    data_lock,          // handle to semaphore
    1,                  // increase count by one
    NULL);              // not interested in previous count
VPRINT("Waiting for thread to join !\n");

VPRINT("Delete the semaphore !\n");
CloseHandle(data_lock);

if (logging) {
    VPRINT("Logging Enabled, flush !\n");
    if (file_flush) {
        fflush(lf);
    }
    if (lf) {
        fclose(lf);
        lf = NULL;
        VPRINT("Log file closed\n");
    }
}

// Sleep(5000);
return exit_value;
}

//-----

int main(int argc, char * argv[]) {

    printf("Berkeley BioScientific - %s\n", VERSION);
    my_main(argc, argv);
    VPRINT("ok .. done\n");
    return 0;
}

```

Appendix E Radio Frequency Spectrometer Acquisition for Windows®

SignalHound SA44B Radio frequency Spectrum Analyzer.

```
// sa44.cpp : Defines the entry point for the console application.
//

#include <windows.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <io.h>
#include <tchar.h>
#include <stdio.h>
#include <time.h>
#include <strsafe.h>
#include <Winerror.h>
#include "mosquitto.h"
#include <process.h>

#include "sa_api.h"

#define VERSION "V1.1.1"

#define SM_BUF 1024
#define MQ_BUF 512*1024

#define SA44_MAX 4096

#define RUNNING 2
#define STOPPED 1

#define LINK_DOWN 0
#define LINK_UP 1

#define KEEPALIVE 60
#define MQTT_PORT 1883

#define F_EXIST 00
#define F_WRITE_PERMISSION 01
#define F_READ_PERMISSION 04
#define F_READ_WRITE_PERMISSION 6

//-----
// Macros

#if 0
#define VPRINT(A)
#define VDPRINT(A)
#else
#define VPRINT(fmt, ...) do { if (flag_v) { printf (fmt, ##__VA_ARGS__); } } while (0)
#define VDPRINT(fmt, ...) do { if (flag_v && flag_D) { printf (fmt, ##__VA_ARGS__); } } while (0)
#endif

//-----

// Infile stuff
char inifile[SM_BUF] = "rfs0.ini";
char *arg_inifile = inifile;

// Logfile stuff
char logfile[SM_BUF] = "RF_logfile";
FILE *lf = NULL;
int logging = 0;
char logday[SM_BUF] = "";
```

```

int file_flush = 0;

// Mosquitto
char sbroker[SM_BUF] = "localhost";
char clientid[SM_BUF];
int mqport;
char topicname[SM_BUF] = "ICE121/RF0";
char sensorname[SM_BUF] = "RF0";
int persist = 0;
int keepalive;
int link_status = LINK_DOWN;

// Thread and Mutex
HANDLE data_lock;
HANDLE mq_worker;
DWORD ThreadID;

// Licence
char licence[SM_BUF];

volatile int keep_going = 1;
volatile int mode = STOPPED; // lock for DO

// usage flags
int flag_t = 0; /* Test mode */
int flag_v = 0; /* verbose */
int flag_D = 0; /* debug */
int flag_i = 0; /* infile .. mandatory */

char devname[SM_BUF];
char serialnumber[SM_BUF];
int spectrometerhandle = -1;

float centrefrequency = 60000000.0;
float spanfrequency = 100000000.0;
float rbwfrequency = 30000.0;
float vbwfrequency = 30000.0;
float level = -10.0;
int rbwshape = SA_RBW_SHAPE_FLATTOP;
int fegain = 0;
int feattenuation = 0;
int fepreamp = 0;
int thresholdenable = 0;
float thresholdvalue = 0.0;

long long total_read_count = 0;

// VC only has 1MB stack .. so make global space
char mq_buf[MQ_BUF];
char header_buf[SM_BUF];
char data_buf[MQ_BUF];
float min_spectrum[SA44_MAX * 16]; // allow for plenty of space ! This is only 64K floats or 256Kbytes
float max_spectrum[SA44_MAX * 16];

//-----
//-----

BOOL WINAPI consoleHandler(DWORD signal) {
    switch (signal) {
        case CTRL_C_EVENT:
            keep_going = false;
            printf("Ctrl-C handled\n"); // do cleanup
            return TRUE;

        default:
            printf("Event [%d] handled\n", signal);
            return FALSE;
    }
}

//-----
//-----

```

```

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n", (void*)&mosq, obj, result);
}

//-----

void message_callback(struct mosquitto *mosq, void *obj,
    const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n", (void*)&mosq,
        (void*)&obj, (void*)&message);
}

//-----
//-----

#pragma comment(lib, "sa_api.lib")

// Selectively open a device by specifying the serial number
// of the target device

int openDeviceBySerial(char *d) {
    int handle = -1;
    int i;
    int serials[8] = { 0 }, deviceCount;
    saStatus openStatus;
    saDeviceType deviceType;
    int sn = 0;
    int res;
    char types[SM_BUF];

    printf("Api Version    --> %s\n", saGetAPIVersion());
    printf("Looking for      --> %s\n", d);

    saGetSerialNumberList(serials, &deviceCount);

    if (deviceCount <= 0) {
        fprintf(stderr, "Error scanning for devices\n");
        return -1;
    }

    printf("Found %d device(s)\n", deviceCount);
    for (i = 0; i < deviceCount; i++) {
        printf("Found device [%d] -> Serial Number -> 0x%.8x \n", i, serials[i]);
    }

    // Find desired serial number
    res = sscanf_s(d, "%x", &sn);
    if (1 != res) {
        fprintf(stderr, "Could not understand serial number specified ->%s\n", d);
        return -1;
    }
    for (i = 0; i < deviceCount; i++) {
        if (sn == serials[i]) {
            printf("Found matching serial number [0x%.8x] at %d\n", sn, i);
            openStatus = saOpenDeviceBySerialNumber(&handle, serials[i]);
            if (saNoError != openStatus) {
                fprintf(stderr, "%s\n", saGetErrorString(openStatus));
                return -2; // failed to open ?
            }
            saGetDeviceType(handle, &deviceType);
            snprintf(types, SM_BUF, "%s -> (%d)", "Unknown", deviceType);

            if (saDeviceTypeSA44 == deviceType) {
                snprintf(types, SM_BUF, "%s", "SA44");
            }
            if (saDeviceTypeSA44B == deviceType) {
                snprintf(types, SM_BUF, "%s", "SA44B");
            }
            if (saDeviceTypeSA124A == deviceType) {
                snprintf(types, SM_BUF, "%s", "SA124A");
            }
        }
    }
}

```

```

        if (saDeviceTypeSA124B == deviceType) {
            snprintf(types, SM_BUF, "%s", "SA124B");
        }

        printf("Returning open device (%s) [%d] with sn [0x%.8x] handle -> %d\n", types, i, sn,
handle);

        // Device is open here
        return handle;
    }
}

fprintf(stderr, "Could not find matching serial number 0x%.8x\n", sn);
return -1;
}

//-----
int closeDevice(int handle) {

    // When finished
    saCloseDevice(handle);
    handle = -1;
    return 0;
}

//-----

void mosquito_error(int er) {
    switch (er) {
        case MOSQ_ERR_CONN_PENDING:
            VPRINT("Mosquitto erro Pending\n");
            break;
        case MOSQ_ERR_SUCCESS:
            VPRINT("Mosquitto erro Success\n");
            break;
        case MOSQ_ERR_NOMEM:
            VPRINT("Mosquitto erro NoMem\n");
            break;
        case MOSQ_ERR_PROTOCOL:
            VPRINT("Mosquitto erro ERR Protocol\n");
            break;
        case MOSQ_ERR_INVALID:
            VPRINT("Mosquitto erro Invalid\n");
            break;
        case MOSQ_ERR_NO_CONN:
            VPRINT("Mosquitto erro No CONN\n");
            break;
        case MOSQ_ERR_CONN_REFUSED:
            VPRINT("Mosquitto erro CONN refused\n");
            break;
        case MOSQ_ERR_NOT_FOUND:
            VPRINT("Mosquitto erro Not found\n");
            break;
        case MOSQ_ERR_CONN_LOST:
            VPRINT("Mosquitto erro CONN lost\n");
            break;
        case MOSQ_ERR_TLS:
            VPRINT("Mosquitto erro TLS\n");
            break;
        case MOSQ_ERR_PAYLOAD_SIZE:
            VPRINT("Mosquitto erro payload size\n");
            break;
        case MOSQ_ERR_NOT_SUPPORTED:
            VPRINT("Mosquitto erro not supported\n");
            break;
        case MOSQ_ERR_AUTH:
        case MOSQ_ERR_ACL_DENIED:
            VPRINT("Mosquitto erro ACL denied\n");
            break;
        case MOSQ_ERR_UNKNOWN:
            VPRINT("Mosquitto erro unknown\n");
            break;
        case MOSQ_ERR_ERRNO:
            VPRINT("Mosquitto connect before loop start\n");
            break;
        case MOSQ_ERR_EAI:
    }
}

```

```

        VPRINT("Mosquitto erro EAI, getinfoaddr\n");
        break;
    default:
        VPRINT("Mosquitto default - no error defined\n");
        break;
    }
}

//-----

char tohex(int chr) {
    chr &= 0x0f;
    switch (chr) {
        case 0x00: return '0'; case 0x01: return '1';
        case 0x02: return '2'; case 0x03: return '3';
        case 0x04: return '4'; case 0x05: return '5';
        case 0x06: return '6'; case 0x07: return '7';
        case 0x08: return '8'; case 0x09: return '9';
        case 0x0a: return 'a'; case 0x0b: return 'b';
        case 0x0c: return 'c'; case 0x0d: return 'd';
        case 0x0e: return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}

//-----

unsigned long long current_timestamp(int flag) {

    ULARGE_INTEGER theTime;
    unsigned __int64 fileTime64Bit;
    FILETIME ft;

    GetSystemTimePreciseAsFileTime(&ft);           // returns highest precision time
    theTime.LowPart = ft.dwLowDateTime;              // number of 100nS intervals since
    theTime.HighPart = ft.dwHighDateTime;            // Jan 1 1601(UTC)
    fileTime64Bit = theTime.QuadPart;

    return (fileTime64Bit);
}

//-----

int wr3_day(char * b, int len) {
    time_t t;
    struct tm tm;
    int wday;
    char *sday[7] = { "Sun", "Mon", "Tue", "Wed", "Thurs", "Fri", "Sat" };

    t = time(NULL);
    localtime_s(&tm, &t);

    wday = tm.tm_wday;
    snprintf(b, len, "%s", sday[wday]);
    return 0;
}

//-----

int wr3_date(char * tim_buf, int len, int dec_places) {

#define D_FORMAT "%A"
#define TDZ_FORMAT "%z"
#define YMDHMS_FORMAT "%Y-%m-%dT%H:%M:%S."
#define BUF 64

    __time64_t ltm;
    struct tm tm;
    errno_t err;
    size_t sl;           // chars returned
    char tdz[BUF];       // timezone
    SYSTEMTIME st;
    int res;

```



```

    _tzset();
    _time64(&ltm);
    err = _localtime64_s(&tm, &ltm);
    GetSystemTime(&st);

    // Get time zone difference as "xxxx"
    // such that can expand to "xx:xx"
    sl = strftime(tdz, sizeof(tdz), TDZ_FORMAT, &tm);
    if (0 == sl
        ) {
        fprintf(stderr, "strftime return 0\n");
        return -2;
    }
    // simple assignation of the 5 bytes for format required
    tdz[6] = 0; // zero terminate
    tdz[5] = tdz[4];
    tdz[4] = tdz[3];
    tdz[3] = ':';
    // printf("Timezone is %s\n", tdz);

    res = snprintf(tim_buf, len,
        //yyy-mm-ddThh:mm:ss.fff+01:00
        "%0.4d-%0.2d-%0.2dT%0.2d:%0.2d:%0.3d%s",
        st.wYear,
        st.wMonth,
        st.wDay,
        st.wHour,
        st.wMinute,
        st.wSecond,
        st.wMilliseconds,
        tdz);

    // printf("OUTBUF is -> '%s'\n", tim_buf);

    return 0;
}
//-----

int build_shaper_mq_header(char *buf,
    int buf_len,
    char *sensor_name,
    char *data_type,
    char *data_structure,
    int elements,
    char *separator,
    int msg_len) {

    int res;
    char tim_buf[SM_BUF];

    res = wr3_date(tim_buf, SM_BUF, 4/* decimal places*);
    if (res) {
        fprintf(stderr, "Error in datetime function\n");
    }

    res = snprintf(buf, buf_len, "%s;%s;%s;%s;%d;%s;%d",
        sensor_name,
        tim_buf,
        data_type,
        data_structure,
        elements,
        separator,
        msg_len);

    if (buf_len < res) {
        fprintf(stderr, "Could not pack header, buf too small %d:%d\n", buf_len, res);
        return -1;
    }
    return res;
}
//-----

```

```

// Calc checksum to insert in 'three digit [000 to 255] checksum and ':'
int calc_tail(char * mq_bf, int mlen) {
    int i;
    int sum = 0;

    for (i = 0; i < mlen - 4; i++) { // exclude already add 'digits' and ':'
        sum += mq_bf[i];
    }
    sum &= 0xff;
    snprintf(&mq_bf[mlen - 4], 4, "%03d", sum); // left justified 3 digits decimal

    return 0;
}

//-----

void usage() {
    puts("Usage: sa44 -c infile\n"
        "Examples:\n"
        "  $ sa44 -c rfs0.ini\n");
    exit(1);
}

//-----

int load_ini_parms(TCHAR * ini_file) {
    DWORD res;
    TCHAR rfsunit[SM_BUF];
    TCHAR _clientid[SM_BUF]; // temp space
    TCHAR sAbsolutePathBuffer[2048] = _T(""); // getprofile requires full pathname !!

    GetFullPathName(ini_file, sizeof(sAbsolutePathBuffer) / sizeof(TCHAR), sAbsolutePathBuffer, NULL);

    // General stuff
    VDPRINT(("Read --> %s\n", "General:Logfile"));
    res = GetPrivateProfileString(_T("General"), _T("Logfile"), _T("OES"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
    _stprintf_s(logfile, MQ_BUF, _T("%s"), rfsunit);

    VDPRINT(("Read --> %s\n", "General:Logging"));
    res = GetPrivateProfileString(_T("General"), _T("Logging"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
    logging = !(_ttoi(rfsunit));

    VDPRINT(("Read --> %s\n", "General:Licence"));
    res = GetPrivateProfileString(_T("General"), _T("Licence"), _T("Not Licensed"), licence, SM_BUF,
sAbsolutePathBuffer);

    VDPRINT(("Read --> %s\n", "General:Flush"));
    res = GetPrivateProfileString(_T("General"), _T("Flush"), _T("0"), rfsunit, SM_BUF, sAbsolutePathBuffer);
    file_flush = !(_atoi(rfsunit));

    // Spectrometer stuff
    VDPRINT(("Read --> %s\n", "Spectrometer:device"));
    res = GetPrivateProfileString(_T("Spectrometer"), _T("device"), _T("NoDevice"), devname, SM_BUF,
sAbsolutePathBuffer);

    VDPRINT(("Read --> %s\n", "Spectrometer:sensorname"));
    res = GetPrivateProfileString(_T("Spectrometer"), _T("sensorname"), _T("OES0"), sensorname, SM_BUF,
sAbsolutePathBuffer);

    VDPRINT(("Read --> %s\n", "Spectrometer:Centrefrequency"));
    res = GetPrivateProfileString(_T("Spectrometer"), _T("Centrefrequency"), _T("60000000.0"), rfsunit,
SM_BUF, sAbsolutePathBuffer);
    res = sscanf_s(rfsunit, "%f", &centrefrequency);
    if (1 != res) {
        centrefrequency = 60000000.0;
        fprintf(stderr, "Could not read Centrefrequency '%s', defaulting to %.2f\n", rfsunit,
centrefrequency);
    }

    VDPRINT(("Read --> %s\n", "Spectrometer:Spanfrequency"));
    res = GetPrivateProfileString(_T("Spectrometer"), _T("Spanfrequency"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);

```

```

res = sscanf_s(rfsunit, "%f", &spanfrequency);
if (1 != res) {
    spanfrequency = 100000000.0;
    fprintf(stderr, "Could not read Spanfrequency '%s', defaulting to %.2f\n", rfsunit, spanfrequency);
}

VDPRI("Read --> %s\n", "Spectrometer:RBW");
res = GetPrivateProfileString(_T("Spectrometer"), _T("RBWfrequency"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
res = sscanf_s(rfsunit, "%f", &rbwfrequency);
if (1 != res) {
    rbwfrequency = 30000.0;
    fprintf(stderr, "Could not read RBWfrequency '%s', defaulting to %.2f\n", rfsunit, rbwfrequency);
}

VDPRI("Read --> %s\n", "Spectrometer:VBW");
res = GetPrivateProfileString(_T("Spectrometer"), _T("VBWfrequency"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
res = sscanf_s(rfsunit, "%f", &vbwfrequency);
if (1 != res) {
    vbwfrequency = 30000.0;
    fprintf(stderr, "Could not read VBWfrequency '%s', defaulting to %.2f\n", rfsunit, vbwfrequency);
}

VDPRI("Read --> %s\n", "Spectrometer:RBWshape");
res = GetPrivateProfileString(_T("Spectrometer"), _T("RBWshape"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
// if ('N' == toupper(rfsunit[0])) { // Nuttall
//     rbwshape = SA_RBW_SHAPE_NUTTALL;
// }
if ('F' == toupper(rfsunit[0])) {
    rbwshape = SA_RBW_SHAPE_FLATTOP;
}
if ('C' == toupper(rfsunit[0])) {
    rbwshape = SA_RBW_SHAPE_CISPR;
}

VDPRI("Read --> %s\n", "Spectrometer:Preamplifier");
res = GetPrivateProfileString(_T("Spectrometer"), _T("Preamplifier"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
fepreamplifier = !!atoi(rfsunit);

VDPRI("Read --> %s\n", "Spectrometer:Attenuation");
res = GetPrivateProfileString(_T("Spectrometer"), _T("Attenuation"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
feattenuation = atoi(rfsunit);

VDPRI("Read --> %s\n", "Spectrometer:Gain");
res = GetPrivateProfileString(_T("Spectrometer"), _T("Gain"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
fegain = atoi(rfsunit);

VDPRI("Read --> %s\n", "Spectrometer:ThresholdEnable");
res = GetPrivateProfileString(_T("Spectrometer"), _T("Thresholdenable"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
thresholdenable = !!atoi(rfsunit);

VDPRI("Read --> %s\n", "Spectrometer:ThresholdValue");
res = GetPrivateProfileString(_T("Spectrometer"), _T("ThresholdValue"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
res = sscanf_s(rfsunit, "%f", &thresholdvalue);
if (1 != res) {
    thresholdvalue = -20.0;
    fprintf(stderr, "Could not read Threshold Value '%s', defaulting to %.2f\n", rfsunit, thresholdvalue);
}

// Mosquito Stuff
VDPRI("Read --> %s\n", "Mosquito:topicname");
res = GetPrivateProfileString(_T("Mosquito"), _T("topicname"), _T("OES"), topicname, SM_BUF,
sAbsolutePathBuffer);

// set up a reasonable random client id (can pass NULL to procedure!)
memset(_clientid, 0, 24); // generate a temporary one using PID
sprintf(_clientid, 23, "oes_%d", GetProcessId(NULL));

```

```

VDPRI("Read --> %s\n", "Mosquitto:clientid");
// only 24 characters permitted for 'client_id' // leave 1 for 'NULL'
res = GetPrivateProfileString(_T("Mosquitto"), _T("clientid"), _clientid, clientid, 23, sAbsolutePathBuffer);

VDPRI("Read --> %s\n", "Mosquitto:broker");
res = GetPrivateProfileString(_T("Mosquitto"), _T("broker"), _T("localhost"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
snprintf(sbroker, SM_BUF, rfsunit);

VDPRI("Read --> %s\n", "Mosquitto:mqport");
res = GetPrivateProfileString(_T("Mosquitto"), _T("mqport"), _T("MQTT_PORT"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
mqport = _ttoi(rfsunit);

VDPRI("Read --> %s\n", "Mosquitto:keepalive");
res = GetPrivateProfileString(_T("Mosquitto"), _T("keepalive"), _T("KEEPALIVE"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
keepalive = atoi(rfsunit);

VDPRI("Read --> %s\n", "Mosquitto:persist");
res = GetPrivateProfileString(_T("Mosquitto"), _T("persist"), _T("0"), rfsunit, SM_BUF,
sAbsolutePathBuffer);
keepalive = !(atoi(rfsunit));

return 0;

}

//-----

void flags_init(int argc, char *argv[]) {
    int i;
    for (i = 1; i<argc; i++) {
        if (!strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (!strcmp(argv[i], "-t")) { flag_t = 1; continue; }
        if (!strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (!strcmp(argv[i], "-c") && (i + 1 < argc)) {
            flag_i = 1; arg_inifile = argv[++i]; continue;
        }

        fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
        usage();
    }

    if (flag_i && arg_inifile) {
        if (strlen(arg_inifile) < sizeof(inifile)) {
            snprintf(inifile, SM_BUF, "%s", arg_inifile);
            if (_access_s(arg_inifile, F_READ_PERMISSION) != -1) {
                // file exists
                VPRINT("Found ini file\n");
            }
            else {
                // file doesn't exist
                fprintf(stderr, "Could find ini file\n");
            }
        }
    }

    if (!flag_i) {
        fprintf(stderr, "You must specify an INI file\n");
        usage(); // does not return
    }

    load_ini_parms(inifile); // read some ini parms

    printf("Inifile is      --> %s\n", inifile);
    printf("Licence is        --> %s\n", licence);
    printf("Broker is         --> %s:%d\n", sbroker, mqport);
    printf("ClientID is       --> %s\n", clientid);
    printf("Topic is          --> %s\n", topicname);
    printf("Sensor is         --> %s\n", sensorname);
    printf("Device is         --> %s\n", devname);
    printf("Debug is          --> %d\n", flag_D);
    printf("Verbose is        --> %d\n", flag_v);
    printf("Test Mode is      --> %d\n", flag_t);

```

```

printf("Logging is      --> %d\n", logging);
printf("Logfile is      --> '%s'\n", logfile);
printf("Logfile Flush is --> %d\n", file_flush);

printf("Centre Frequency --> %f\n", centrefrequency);
printf("Span Frequency   --> %f\n", spanfrequency);
printf("RBW Frequency    --> %f\n", rbwfrequency);
printf("VBW Frequency    --> %f\n", vbwfrequency);
printf("RBW shape        --> %s\n", (SA_RBW_SHAPE_FLATTOP == rbwshape) ? "FLATTOP" :
      (SA_RBW_SHAPE_CISPR == rbwshape) ? "CISPR" : "Unknown");
printf("Reference level  --> %f\n", level);
printf("Gain             --> %d\n", fegain);
printf("Attenuation       --> %d\n", feattenuation);
printf("PreAmp           --> %d\n", fepreamp);
printf("Threshold enabled --> %s\n", thresholdenable ? "Yes" : "No");
printf("Threshold value   --> %f\n", thresholdvalue);

fflush(NULL);
}

//-----
//-----

// Only does a 'loop' when semaphore is set from main
// which reads the data from the SA44
//
static void * post_thread(void *arg) {

    char *user_ptr = NULL;
    char svalue[32];

    int res,res1;
    int dl; // data length
    int ml; // message length
    int i;

    int error = 0;
    int sweepLen;
    double startFreq;
    double binSize;
    float peak_spectrum = 0.0;

    saStatus sastatus;

    struct mosquito *mosq = NULL;

    VPRINT(">>Entered Worker thread with argp -> %p\n", arg);

    mosquito_lib_init();
    mosq = mosquito_new(clientid, 1, user_ptr);
    if (mosq) {
        mosquito_connect_callback_set(mosq, connect_callback);
        mosquito_message_callback_set(mosq, message_callback);
        res = mosquito_connect(mosq, sbroker, mqport, keepalive);
        if (res) {
            mosquito_error(res);
            printf("%s Link Down %d\n", sensorname, res);
            link_status = LINK_DOWN;
        }
        else {
            printf("%s Link UP\n", sensorname);
            link_status = LINK_UP;
        }
    }
    else {
        fprintf(stderr, "Could not start Mosquitto\n");
        exit(1);
    }
    // start network loop thread

```

```

mosquitto_loop_start(mosq);

saConfigCenterSpan(spectrometerhandle, centrefrequency, spanfrequency);
saConfigAcquisition(spectrometerhandle, SA_MIN_MAX, SA_LOG_SCALE);
saConfigLevel(spectrometerhandle, level);
saConfigSweepCoupling(spectrometerhandle, rbwfrequency, vbwfrequency, true);

// Initialize the device with the configuration just set
sastatus = saInitiate(spectrometerhandle, SA_SWEEPING, 0);
if (saNoError != sastatus) {
    // Handle unable to initialize
    fprintf(stderr, "Failed to initiate Spectrometer -> %s\n", saGetErrorString(sastatus));
    return NULL;
}
saQuerySweepInfo(spectrometerhandle, &sweepLen, &startFreq, &binSize);
printf("Start Frequency -> %f\n", startFreq);
printf("Sweep Length   -> %d\n", sweepLen);
printf("Bin size      -> %f\n", binSize);

// now main loop .. just wait semaphore to run and acquire sample
while (keep_going) {
    res = WaitForSingleObject(data_lock, INFINITE); // down semaphore, data is in

    // potential race here on data, but

    // serial data is 'relatively' slow

    // we dont 'critical section' the data

    // as this may 'hold' up serial processing
    if (0 != res) {
        res1 = GetLastError();
        char buf[SM_BUF];
        FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
NULL,
res1, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
buf,
sizeof(buf),
NULL);
        fprintf(stderr, "Error waiting semaphore %d:%d '%s'\n", res, res1, buf);
    }
    if (keep_going) { // check one
more time, remove that last one !

        sastatus = saGetSweep_32f(spectrometerhandle, min_spectrum, max_spectrum);

        if (saNoError != sastatus) {
            fprintf(stderr, "Failed to get spectrum -> %s\n", saGetErrorString(sastatus));
        }

        data_buf[0] = 0;
        for (i = 0; i < sweepLen - 1; i++) {
            snprintf(svalue, 32, "%.3f", max_spectrum[i]);
            strcat_s(data_buf, MQ_BUF, svalue);
        }
        snprintf(svalue, 32, "%.3f", max_spectrum[sweepLen - 1]); // final one .. no ','
        strcat_s(data_buf, MQ_BUF, svalue);
        dl = strlen(data_buf, MQ_BUF);

#if 0
        for (i = 0; i < 10 /*pixels_read*/; i++) {
            printf("%d -> %.2f ", i, max_spectrum[i]);
            // dl += snprintf(&data_buf[i], MQ_BUF, "%f", max_spectrum[i]);
        }
        printf("\n");
        for (i = 1000; i < 1000 + 10 /*pixels_read*/; i++) {
            printf("%d -> %.2f ", i, max_spectrum[i]);
            // dl += snprintf(&data_buf[i], MQ_BUF, "%f", max_spectrum[i]);
        }
        printf("\n");

```

#endif

```
if (thresholdenable) {
    peak_spectrum = max_spectrum[0];
    for (i = 0; i < sweepLen; i++) {
        if (peak_spectrum < max_spectrum[i]) {
            peak_spectrum = max_spectrum[i];
        }
    }
    if (thresholdvalue > peak_spectrum) { // didnt meet the requirement
        goto skip;
    }
}
build_shaper_mq_header(header_buf,
    SM_BUF,
    sensorname,
    "CSV",
    "float",
    28,
    /* "\",\", *, */
    ", ",
    dl);

ml = snprintf(mq_buf, MQ_BUF, "%s|%s|%03d|", header_buf, data_buf, 0xff);
calc_tail(mq_buf, ml);

switch (link_status) {
case LINK_UP:
    res = mosquitto_publish(mosq,
        NULL,
        topicname,
        strlen(mq_buf),
        mq_buf,
        0,
        0);

    if (res) { // not good !
        link_status = LINK_DOWN;
        printf("%s -> link down \n", topicname);
    }
    break;

case LINK_DOWN:
    res = mosquitto_reconnect(mosq); // broker down, attempt reconnect
    if (!res) {
        link_status = LINK_UP;
        printf("%s -> link up \n", topicname);
    }
    break;

default:
    fprintf(stderr, "%s Internal error on link status %s\n", topicname, sbroker);
    break;
} // link_status ?

if (flag_D) {
    printf("%s:%s\n", sensorname, mq_buf);
}

} // keep_going one last time
skip:
    ; // if thresholding enabled and we didnt meet requirement
} // while keep_going

mosquitto_disconnect(mosq);
mosquitto_destroy(mosq);
VPRINT("%s Disconnected from broker %s\n", topicname, sbroker);

return NULL;
}
```

```
//-----

int my_main(int argc, char **argv) {

    char *desired_type = NULL;          // what spectrometer model we want
    char *desired_serial_number = NULL;  // what spectrometer SN we want
    int c = 0;
    int res, res1;
    int exit_value = EXIT_SUCCESS;
    long long unsigned start_time, stop_time, delta_time;
    int loop_read_count = 0;
    float temperature;
    float voltage;
    int loop_max = 10; // once per second

    //-----

    flags_init(argc, argv);
    desired_serial_number = devname;

    start_time = current_timestamp(0);          // hundreds of nanoseconds since jan 1 1601
    printf("Time starting is %lld\n", start_time);

    fprintf(stderr, "Finding connected radio spectrometers...\n");

    spectrometerhandle = openDeviceBySerial(desired_serial_number);
    if (spectrometerhandle < 0) {
        fprintf(stderr, "Failed to open Spectrometer [%s]\n", desired_serial_number);
        goto done;
    }
    else {
        printf("Found spectrometer -> %d\n", spectrometerhandle);
    }

    // get temperature
    saQueryTemperature(spectrometerhandle, &temperature); // will do this periodically
    saQueryDiagnostics(spectrometerhandle, &voltage);

    printf("Temperature -> %.2f degC, Voltage -> %.2fV\n", temperature, voltage);

    data_lock = CreateSemaphore(
        NULL,          // default security attributes
        0,             // Initial Count
        1,             // maximum count
        NULL);         // unnamed semaphore

    if (NULL == data_lock) {
        fprintf(stderr, "Could not create semaphore\n");
        exit_value = EXIT_FAILURE;
        goto done;
    }

    mq_worker = CreateThread(
        NULL,          // default security attributes
        0,             // default stack size
        (LPTHREAD_START_ROUTINE)post_thread,
        NULL,          // No tread functions arguments
        0,             // default creation flags
        &ThreadID);    // receive thread identifier

    keep_going = TRUE;

    if (!SetConsoleCtrlHandler(consoleHandler, TRUE)) {
        fprintf(stderr, "Could not set control handler\n");
        exit_value = EXIT_FAILURE;
        goto done;
    }
}
```



```

while (keep_going) {

    Sleep(100); // wait loop_max * THIS
    total_read_count++;
    loop_read_count++;

    if (loop_read_count >= loop_max) { // time to estimate ?
        loop_read_count = 0;

        res = ReleaseSemaphore(
            data_lock,                // handle to semaphore
            1,                        // increase count by one
            NULL);                    // not interested in previous count

        if (!res) {
            res1 = GetLastError();
            char buf[SM_BUF];
            FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
                        NULL,
                        res1, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                        buf,
                        sizeof(buf),
                        NULL);
            fprintf(stderr, "Error bumping semaphore %d:%d '%s'\n", res, res1, buf);
            fprintf(stderr, "Suggest sweep is taking too long .. Adjust the Sweep
Paramaters\n");

            loop_max += 5;
        }
        else {
            if (loop_max > 10) {
                loop_max -= 5;
                fprintf(stderr, "Set loop_max to %d\n", loop_max);
            }
        }
    }

    } // want to decimate the data_rate !
} // end while keep-going

done:

keep_going = 0; // force for debug, if we are jumping here

printf("Ok .. Now closing device with handle -> %d\n", spectrometerhandle);
closeDevice(spectrometerhandle);
printf("Ok .. Now closed device with handle -> %d\n", spectrometerhandle);
spectrometerhandle = -1;

VPRINT("No of records read was : %lld\n", total_read_count);

stop_time = current_timestamp(0);                // hundreds of nanoseconds since jan 1 1601

printf("Time start      :      is %llu (100nS)\n", start_time);
printf("Time now       :      is %llu (100nS)\n", stop_time);

delta_time = stop_time - start_time;
printf("Delta time      :      is %llu (mS)\n", delta_time / (10 * 1000));
printf("Total loop count :      is %lld\n", total_read_count);

res = ReleaseSemaphore( // send one to free it !
    data_lock,          // handle to semaphore
    1,                  // increase count by one
    NULL);              // not interested in previous count
VPRINT("Waiting for thread to join !\n");

VPRINT("Delete the semaphore !\n");
CloseHandle(data_lock);

if (logging) {
    VPRINT("Logging Enabled, flush !\n");

```

```

        if (file_flush) {
            fflush(lf);
        }
        if (lf) {
            fclose(lf);
            lf = NULL;
            VPRINT("Log file closed\n");
        }
    }

    // Sleep(5000);
    return exit_value;
}

//-----

int main(int argc, char * argv[]) {

    printf("Berkeley BioScientific - %s\n", VERSION);
    my_main(argc, argv);
    VPRINT("ok .. done\n");
    return 0;
}

```

Appendix F SquareLaw Demodulator

Oxford 100 Square Law for Matlab®

```
% Get the common filename
filename = 'e_b.csv';
% Pick up the E-Field and the B-Field .
% The columns are
% time(nS), E, B
% 2nd row, take all columns (top left is 0,0)
teb=csvread(filename,1,0);
%
% Separate out the columns
%
t = teb(:,1);
e = teb(:,2);
b = teb(:,3);
%
% Every Row entry in the file is a 2nS sample
% The time series runs from 0 to 2798nS
% There are 1400 samples
% samples = time (2798) /2 + 1 = 1400
% sample frequency = 2nS = 500MHz (from time stamps in file)
Fs = 500000000;
%
N=length(t);
%
DownsampleFactor = 15;

figure(1);
plot(t,b);
title('B-Field Antenna measurement');
xlabel('Time (ns)');
ylabel('Volts (V)');
grid;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get envelope detection using Square Detector

sigsq = 2 * b .* b;
% plot it
%plot (t, sigsq);
%hold on;
%plot (t, b);
%hold off;
% Get a low pass filter for the envelope
d2 = fdesign.lowpass('N,Fc',400,10e6,500e6);
designmethods(d2)
Hd2 = design(d2); % 'Window' is the only design method
fvtool(Hd2)
% Do we want to map the response of this filter ?
%measure(Hd2)
% Now use the filter to filter the Square o/p
bsquare_lp = filter(Hd2,sigsq);
%
% We get the square root to get the envelope
```

```
demodb = sqrt(bsquare_lp);  
% We need to delay the B signal by the filter delay to plot  
plot (t, b);  
hold on;  
plot(t, real(demodb));  
hold off
```

Appendix G Hilbert Transform Demodulator

Oxford 100 Hilbert Transform for Matlab®

```
% Get the common filename
filename = 'e_b.csv';
% Pick up the E-Field and the B-Field .
% The columns are
% time(nS), E, B
% 2nd row, take all columns (top left is 0,0)
teb=csvread(filename,1,0);
%
% Separate out the columns
%
t = teb(:,1);
e = teb(:,2);
b = teb(:,3);
%
% Every Row entry in the file is a 2nS sample
% The time series runs from 0 to 2798nS
% There are 1400 samples
% samples = time (2798) /2 + 1 = 1400
% sample frequency = 2nS = 500MHz (from time stamps in file)
Fs = 500000000;
%
N=length(t);
%

%figure(1);
%plot(t,b);
%title('B-Field Antenna measurement');
%xlabel('Time (ns)');
%ylabel('Volts (V)');
%grid;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get envelope detection using Hilbert Detector
% normally you bandpass before hilbert as (noisy) complex phase is
produced
% you would want a narrowband signal, but lets proceed
% on acquired data
f = fft(b);
% create a copy that is multiplied by the complex operator
complexf = 1i*f;

% find indices of positive and negative freuqeucies
posF = 2:floor(N/2)+mod(N,2);
negF = ceil(N/2)+1+-mod(N,2):N;

%rotate fourier coeffiecients
% this computes the i Axis(2pft) component .. ie phase quadrature
% the positive frequencies are rotated counter clockwise
% negative frequencies are rotated clockwise
f(posF) = f(posF) + -1i*complexf(posF);
f(negF) = f(negF) + 1i*complexf(negF);

% now take the inverse fourier to calculate the Hilbert
hilbertf = ifft(f);
```

```

% The simple method is to use the 'toolbox' hilbert function
hilbert_int = hilbert(b);

% lets plot the results
figure(1), clf
subplot(311)
plot(abs(hilbert_int))
hold on
plot(abs(hilbertf))
set(gca, 'xlim', [.5,N+.5])
legend({'Matlab Hilbert function'; '"basic function Hilbert"'})
title('Hilbert Transform')

% lets get the signal plot
subplot(312)
plot(t,real(hilbert_int)), hold on
legend({'real(hilbert(b));'})
xlabel('Time nS'), ylabel('Voltage');

% lets get the phase plot
subplot(313)
plot(t,b), hold on
plot(t,angle(hilbert_int), 'r')
legend({'angle(hilbert(b));'})
xlabel('Time nS'), ylabel('radians');

% 3D plot to show spiral progression of phase
figure(2), clf
plot3(t,real(hilbert_int), imag(hilbert_int))
xlabel('time nS'), ylabel('real part'), zlabel('imag part')
axis tight
rotate3d

```

Appendix H Continuous and Discrete Wavelet Demodulator

Oxford Instruments PlasmaLab 100 CCP Tool Wavelet Transform for Matlab®

```
% Get the common filename
filename = 'e_b.csv';
% Pick up the E-Field and the B-Field .
% The columns are
% time(nS), E, B
% 2nd row, take all columns (top left is 0,0)
teb=csvread(filename,1,0);
%
% Separate out the columns
%
t = teb(:,1);
e = teb(:,2);
b = teb(:,3);
%
% Every Row entry in the file is a 2nS sample
% The time series runs from 0 to 2798nS
% There are 1400 samples
% samples = time (2798) /2 + 1 = 1400
% sample frequency = 2nS = 500MHz (from time stamps in file)
Fs = 500000000;
%
N=length(t);

figure;
spectrogram(b,[], [], [],Fs,'yaxis');

% Try a shorter window
spectrogram(b,128,[], [],Fs,'yaxis'); title('Spectrogram using short
window');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get using Wavelet Transform

figure;
cwt(b,Fs);

% optional fine scale, Number of Voices per octave, Number of Octaves
No = 7;
Nv = 32;
figure;
cwt(b, Fs, 'NumOctaves', No, 'VoicesPerOctave',Nv);
title('CWT with number of Octaves: 7 and Voices per Octave: 32');

% Reconstruct the b field
[wt,f] = cwt(b,Fs);
% reconstruct around the frequencies of interest (1MHz to 150MHz)
brec = icwt(wt, f, [10000000.0, 150000000.0], 'SignalMean', mean(b));
figure;
hold on;

plot(t,brec,'b');
title('Original and Wavelet Reconstructed B field Signal');
```

```
xlabel('Time nS'), ylabel('Voltage');  
axis tight; grid on;  
hold on;  
plot(t,b,'r');
```


Appendix I Software Pattern Matching Engine (PME) Implementation and Results

Coding for the software implementation of the Pattern Matching Engine (PME). This is a software emulation of the architecture of the Intel Quark PME written so as to test the KNN and RBF hyperspace search engines for neural classification of the outputs of the filters from the Discrete Wavelet transform for the detection of Arc events.

```
// pme.c

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <assert.h>
#include <time.h>
// #include <sys/time.h>
#include <math.h>

#include "pme.h"

//-----
// Global flag for exit

volatile int shouldRun = 1;

// Classifier vector sect , arbitrary

unsigned char initial_data [VECTOR_LEN_128] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x38, 0x39, 0x3a, 0x3b, 0x4c, 0x5d, 0x6e, 0x7f,
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x38, 0x39, 0x3a, 0x3b, 0x4c, 0x5d, 0x6e, 0x7f
};

//-----

int comp (const void * elem1, const void * elem2) {
    int f = *((int*)elem1);
    int s = *((int*)elem2);
    if (f > s) return 1;
    if (f < s) return -1;
    return 0;
}

//-----
```

```

void sig_handler(int signo) {
    if (signo == SIGINT) {
        printf("\nSig_handler .. shutting down, please wait\n");
        shouldRun = 0;
    }
}

//-----

void dump_vector(vector128_t *v, int len) {
    int i;
    int j;

    printf("----- Dump Vector ----- \n");
    for (i=0; i<16; i++){
        printf("0x%.2x - ", i*8);

        for (j=0; j<8; j++) {
            printf("0x%.2x ", v->d[i*8 +j]);
        }
        printf("\t");

        for (j=0; j<8; j++) {
            printf("%-2c", v->d[i*8 +j]<32?'*':v->d[i*8 +j]);
        }

        printf("\n");
    }
    printf("----- \n");
}

//-----

void dump_distance(int *d) {
    int i;
    int j;

    printf("----- Dump Distance ----- \n");
    for (i=0; i<16; i++){
        printf("%3d - ", i*8);

        for (j=0; j<8; j++) {
            printf("%14d ", d[i*8 +j]);
        }

        printf("\n");
    }
    printf("----- \n");
}

//-----

void load_vector_set(vector128_t *v) {
    memcpy(&v->d[0], initial_data, sizeof(initial_data));
}

//-----

long get_current_time_with_ms (void) {
    long      ms; // Milliseconds
    time_t    s; // Seconds
    struct timespec spec;

    clock_gettime(CLOCK_REALTIME, &spec);

    s = spec.tv_sec;
    ms = s*1000 + round(spec.tv_nsec / 1.0e6); // Convert nS to mS

    return(ms);
}

```

```

        /*
        printf("Current time: %PRIdMAX.%03ld seconds since the Epoch\n",
               spec, (intmax_t)s, ms);
        */
    }

//-----

int calc_distance(vector128_t *u, vector128_t *v) {
    int i;
    int d2 = 0;
    int d;
    for (i=0; i<128; i++) {
        d = u->d[i] - v->d[i];
        d2 += (d*d);
    }
    return d2;
}

//-----

int find_min_max_128(int *dist, int *min, int *mini, int *max, int *maxi) {
    int i;
    *min = dist[0];
    *max = dist[0];
    *mini = 0;
    *maxi = 0;

    for(i=1; i<128; i++){
        if (dist[i] < *min) {
            *min = dist[i];
            *mini = i;
        }
        if (dist[i] > *max) {
            *max = dist[i];
            *maxi = i;
        }
    }

    return 0;
}

//-----

int main(int argc, char **argv) {

    long start, stop, diff;
    vector128_t pme_set[VECTOR_LEN_128];
    vector128_t to_be_classified;
    int distance[VECTOR_LEN_128];
    int qdistance[VECTOR_LEN_128];
    int i;
    int j;
    int min, mini;
    int max, maxi;

    printf("Version is " VERSION "\n");
    printf("Compiled on " __DATE__ " : " __TIME__ " \n");

    if (SIG_ERR == signal(SIGINT, sig_handler)) {
        perror("Unable to catch SIGINT\n");
        fprintf(stderr, "\ncan't catch SIGINT\n");
    }

    // lets set vector 20 for test
    load_vector_set(&pme_set[20]);

    load_vector_set(&to_be_classified);

    dump_vector(&to_be_classified, VECTOR_LEN_128);

    start = get_current_time_with_ms();

```

```

    for (j=0; j< 1000; j++){
        for (i=0; i<128; i++) {
            distance[i] = calc_distance(&to_be_classified, &pme_set[i]);
        }
        find_min_max_128(distance, &min, &mini, &max, &maxi);
    }
    memcpy(qdistance, distance, sizeof(int) * VECTOR_LEN_128);
    qsort(qdistance, sizeof(qdistance)/sizeof(*qdistance), sizeof(*qdistance), comp);

    stop = get_current_time_with_ms();
    diff = stop - start;

    dump_distance(&distance[0]);
    dump_distance(&qdistance[0]);

    printf("The minimum distance value was    -> %d\n",min);
    printf("The minimum distance value was at  -> %d\n",mini);
    printf("The maximum distance value was    -> %d\n",max);
    printf("The maximum distance value was at  -> %d\n",maxi);
    printf("The time taken for the calc      -> %ld (mS)\n", diff);

    printf("Done\n");

    return 0;
}

//-----

```

// pme.h

```

#ifndef __PME_INCLUDE__
#define __PME_INCLUDE__

#define VECTOR_LEN_128 128

#define BIT(n) (1<<n)

#define VERSION "1.1"

typedef struct vector128 {
    unsigned char d[VECTOR_LEN_128];
} vector128_t;

#endif

```

```

dave@iot: ~/Documents/system_studio/workspace/pme/Debug
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$
dave@iot:~/Documents/system_studio/workspace/pme/Debug$ ./pme
Version is 1.1
Compiled on Jan 3 2019 : 14:48:26
----- Dump Vector -----
0x00 - 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 + + + + + + + +
0x08 - 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f + + + + + + + +
0x10 - 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 + + + + + + + +
0x18 - 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f + + + + + + + +
0x20 - 0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 ! " # $ % & '
0x28 - 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f ( ) * + , - . /
0x30 - 0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0 1 2 3 4 5 6 7
0x38 - 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f 8 9 : ; L | n
0x40 - 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 + + + + + + + +
0x48 - 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f + + + + + + + +
0x50 - 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 + + + + + + + +
0x58 - 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f + + + + + + + +
0x60 - 0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 ! " # $ % & '
0x68 - 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f ( ) * + , - . /
0x70 - 0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0 1 2 3 4 5 6 7
0x78 - 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f 8 9 : ; L | n
----- Dump Distance -----
0 - 275648 275648 275648 275648 275648 275648 275648
8 - 275648 275648 275648 275648 275648 275648 275648
16 - 275648 275648 275648 275648 275648 275648 275648
24 - 275648 275648 275648 275648 275648 275648 275648
32 - 275648 275648 275648 275648 275648 275648 275648
40 - 275648 275648 275648 275648 275648 275648 275648
48 - 275648 275648 275648 275648 275648 275648 275648
56 - 275648 275648 275648 275648 275648 275648 275648
64 - 275648 275648 275648 275648 275648 275648 275648
72 - 275648 275648 275648 275648 275648 275648 275648
80 - 275648 275648 275648 275648 275648 275648 275648
88 - 275648 275648 275648 275648 275648 275648 275648
96 - 275648 275648 275648 275648 275648 275648 275648
104 - 275648 707276 611082 595287 910720 1224900 591263
112 - 768304 590040 362074 565935 568086 1311613 1209161
120 - 1306738 875964 1066533 1170535 508380 275648 1170852
----- Dump Distance -----
0 - 0 272650 275648 275648 275648 275648 275648
8 - 275648 275648 275648 275648 275648 275648 275648
16 - 275648 275648 275648 275648 275648 275648 275648
24 - 275648 275648 275648 275648 275648 275648 275648
32 - 275648 275648 275648 275648 275648 275648 275648
40 - 275648 275648 275648 275648 275648 275648 275648
48 - 275648 275648 275648 275648 275648 275648 275648
56 - 275648 275648 275648 275648 275648 275648 275648
64 - 275648 275648 275648 275648 275648 275648 275648
72 - 275648 275648 275648 275648 275648 275648 275648
80 - 275648 275648 275648 275648 275648 275648 275648
88 - 275648 275648 275648 275648 275648 275648 275648
96 - 275648 275648 275648 275648 275648 275648 275648
104 - 275648 275648 275648 275648 275648 275648 275648
112 - 591263 595287 611082 362074 508380 565935 568086
120 - 1006305 1066533 1170535 1170852 1209161 1224900 1306738
-----
The minimum distance value was -> 0
The minimum distance value was at -> 20
The maximum distance value was -> 1311613
The maximum distance value was at -> 117
The time taken for the calc -> 83 (mS)
Done
dave@iot:~/Documents/system_studio/workspace/pme/Debug$

```

Appendix J Software For Environmental Monitoring of Plasma Systems

Code for monitoring environmental conditions for publishing on the SHAPer MQ protocol message bus.

```
// #includes here
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <pthread.h>
#include <signal.h>
#include <sys/time.h>
#include <errno.h>
#include <netdb.h>
#include "environ.h"
#include "ftd2xx.h"

#include <mosquitto.h>

// # defines etc

#define MQTT_PORT 1883
#define LOCALHOST 0x7f000001
#define LINK_DOWN 0
#define LINK_UP 1
#define NUM_DEV 4

#define MQ_BUF 1024
#define SM_BUF 256

#define MQ_STR "'%s', cast('%s' as datetime),%.1f,%.1f,%.2f"

// globals
char serial_number[SM_BUF] = "A900XTI2"; // we only 'open' the first one !
char myhost[SM_BUF] = "localhost";
char myport[SM_BUF] = "1883";
char mymq_queue[SM_BUF] = "EN";
int baud_rate = 9600;
char delimiter = '*';
char input_buffer[MQ_BUF];
char output_buffer[MQ_BUF];
char *arg_host = myhost;
char *arg_port = myport;
char *arg_ratio = NULL;
char *user_ptr = NULL;
char *arg_queue = mymq_queue;
char *arg_dev = serial_number;
int host = LOCALHOST;
int port = MQTT_PORT;
int link_status = LINK_DOWN;

// FT driver stuff !
FT_STATUS weather_status;
FT_HANDLE weather_handle = NULL;

// weather values from Sparkfun board
float temperature = 0.0; // expect board to be set as degC
int humidity = 0; // as percentage
float dewpoint = 0.0; // temp in degC
float barometric = 0.0; // hectoPascals
float relative_light = 0.0; // amount of light on the light sensor
```

```

float battery = 0.0;    // battery powering device (currently USB powered)

volatile int keepRunning = 1;

// usage flags
int flag_h = 0;          /* host */
int flag_p = 0;          /* port */
int flag_v = 0;          /* verbose */
int flag_n = 0;          /* my queue name */
int flag_d = 0;          /* device name */
int flag_D = 0;          /* debug */

// constants

// some enviromental data !
// the string fro the device is configurable, but
// generally follows
// $,yy.y,jj,a.a,xxxx.xx,z.z,w.www,*<CR><LF>
// where
// $    = header
// yy.y  = temp from humidity sensor
// jj    = humidity percent
// a.a   = dewpoint in degrees C
// xxx.xx = barometric pressure in hPascals
// z.z   = relative light
// w.www  = battery level volts
// *     = record terminator

//-----
#define LO 0
#define HI 1

char tohex(unsigned char chr, int hilo) {
    if (hilo==HI)
        chr = (chr>>4) & 0x0f;
    else
        chr = chr & 0x0f;
    switch(chr) {
        case 0x0: return '0'; case 0x01: return '1'; case 0x02 : return '2'; case 0x03: return '3';
        case 0x4: return '4'; case 0x05: return '5'; case 0x06 : return '6'; case 0x07: return '7';
        case 0x8: return '8'; case 0x09: return '9'; case 0x0a : return 'a'; case 0x0b: return 'b';
        case 0xc: return 'c'; case 0x0d: return 'd'; case 0x0e : return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}
//-----

long long current_timestamp() {
    int res;
    struct timespec ts;
    res = clock_gettime(CLOCK_REALTIME, &ts);
    if (res) {
        perror("could not read clock");
        strerror(errno);
    }
    long long milliseconds = ts.tv_sec*1000LL + ts.tv_nsec/1000000; // calc ms
    return milliseconds;
}

//-----

int tstring(char *buf, int len) {
    int res;
    struct timespec ts;
    struct tm t;

    res = clock_gettime(CLOCK_REALTIME, &ts);
    if (res) {
        perror("could not read clock");
        strerror(errno);
    }

    tzset();

```

```

    if (localtime_r(&(ts.tv_sec), &t) == NULL) {
        return 1;
    }
    res = strftime(buf, len, "%F %T", &t);

    if (0 == res) {
        return 2;
    }
    len -= res - 1;

    // nS reporting
    // res = snprintf(&buf[strlen(buf)], len, "%.09ld", ts.tv_nsec);

    // want mS reporting
    res = snprintf(&buf[strlen(buf)], len, "%.03ld", (ts.tv_nsec/1000000));

    if (res >= len) {
        return 3;
    }

    return 0;
}

//-----

int check_devices(void) {
    FT_STATUS ftStatus;
    char *bufptr[NUM_DEV + 1];
    char sn[NUM_DEV][64]; // space for serial number
    DWORD numdevs = 0;
    int i, res;

    // set up an array of pointers to string bufs for call to ListDevices
    // for API .. last one is NULL
    for(i = 0; i < NUM_DEV; i++) {
        bufptr[i] = sn[i];
        sprintf(sn[i], "%s", serial_number);
    }
    bufptr[NUM_DEV] = NULL;

    ftStatus = FT_ListDevices(bufptr, &numdevs, FT_LIST_ALL | FT_OPEN_BY_SERIAL_NUMBER);
    printf("/-----\n");

    if (ftStatus == FT_OK) {
        if (!numdevs) {
            printf("/ No device\n");
            return -1;
        }
        else {
            for (i = 0; i < (int)numdevs; i++) {
                printf("/ Device -> %d = %s\n", i, bufptr[i]);
            }
            // try opening the particular one that we are interested in !!
            // The user specified the particular number.. so we will index
            // into it !
            res = strcmp(bufptr[i], serial_number, min(sizeof(bufptr[i]), sizeof(serial_number)));
            if (!res) {
                printf("/ Found a Environment station ! %d,%s\n", i, bufptr[i]);
            }
        }
    }
    else {
        fprintf(stderr, "Error getting list of devices\n");
    }
    printf("/-----\n");
    return 0;
}

//-----

int setup_device(void) {
    DWORD dwBytesWritten;
    retry_setup:
    if (!keepRunning) { // just in case no device, and SIGINT called
        return -1;
    }
}

```



```

// lets just check first whether we re-entered on 'retry'
if (weather_handle) {
    FT_Close(weather_handle);
    if (flag_v) {
        printf("%s Closed device %s\n",mymq_queue, serial_number);
    }
    weather_handle = NULL; // mark that it is closed
}

weather_status = FT_OpenEx(serial_number,
    FT_OPEN_BY_SERIAL_NUMBER, &weather_handle);

if (weather_status != FT_OK){
    /*
        This can fail if the ftdi_sio driver is loaded
        use lsmod to check this and rmmod ftdi_sio to remove
        also rmmod usbserial
    */

    if (flag_v) {
        printf("%s Failed to open [%d] .. retry\n",mymq_queue, (int)weather_status);
    }
    sleep(5);
    goto retry_setup;
}
weather_status = FT_SetBaudRate(weather_handle, 9600);
if ((weather_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set baudrate [%d]\n",mymq_queue, (int)weather_status);
    }
    sleep(1);
    goto retry_setup;
}
weather_status = FT_SetTimeouts(weather_handle,5000,0);
if ((weather_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set timeouts [%d]\n",mymq_queue, (int)weather_status);
    }
    sleep(1);
    goto retry_setup;
}
// wake the board up .. send cr !
weather_status = FT_Write(weather_handle, "\n", 1, &dwBytesWritten);
if ((weather_status != FT_OK) || (dwBytesWritten != 1)) {
    if (flag_v) {
        printf("%s Failed to wake board up [%d]\n",mymq_queue, (int)weather_status);
    }
    sleep(1);
    goto retry_setup;
}
printf("%s Device UP\n",mymq_queue);
return(0);
}
//-----
int process_msg() {
    // We stay here and only proceed if the device is online
    // and we can read/assemble a message
    // In all cases of error, we just wait for the
    // device to come online again as there is not much we can do without
    // it !
    char header;
    char trailer;
    int ptr;
    int timeout;
    int res;
    DWORD dwBytesRead;

retry_process:

    if (!weather_handle) { // The device is up and running
        res = setup_device();
        if (res)
            return -1;
    }
}

```

```

if (flag_v) {
    printf("\n%s Waiting\n",mymq_queue);
}
ptr = 0; timeout = 0; // setup for first run
memset(input_buffer, 0, sizeof(input_buffer)); // clear out rubbish

while ((input_buffer[0] != '$') && (input_buffer[ptr] != '*') && keepRunning) { // no record !
    while ((timeout < 2) && keepRunning) { // 2 times 5000 mS --> ~10 seconds
// get one char or times out after 5000mS
        weather_status = FT_Read(weather_handle, &input_buffer[ptr], 1, &dwBytesRead);
        if ((weather_status == FT_OK) && (dwBytesRead == 1)) {
            if (flag_v) {
                printf("%s Got char %d:[%c%c]\n",mymq_queue,ptr,
                    tohex(input_buffer[ptr],HI),
                    tohex(input_buffer[ptr],LO));
            }
            if (input_buffer[ptr] == LF) { // found end
                input_buffer[ptr+1] = 0; // just to be sure, but memset above!!
                goto input_end;
            }
            ptr++; // else general char
        }
        else { // we got timeout or other error, just try again
            fflush(NULL);
            timeout++;
            memset(input_buffer, 0, sizeof(input_buffer)); // dump all before
            ptr = 0;
        }
    }
    if (timeout == 2) { // have we run out of timeouts ??
        if (flag_v) {
            printf("\n%s Oops .. timeout on input\n",mymq_queue);
        }
        sleep(1);
        FT_Close(weather_handle);
        weather_handle = NULL;
        printf("%s Device DOWN\n",mymq_queue);
        goto retry_process;
    }
}
input_end:
// ptr is length -> points to delimiter
// quick check on format
if ((input_buffer[0] == '$') &&
    (input_buffer[ptr-2] == '*') &&
    (input_buffer[ptr-1] == CR) &&
    (input_buffer[ptr] == LF)) { // look correct ?

    input_buffer[ptr-1] = 0; // make a null terminated string
// extract the bits
    if (8!= (res=sscanf(input_buffer,"%1c,%f,%d,%f,%f,%f,%f,%1c",
        &header,
        &temperature,
        &humidity,
        &dewpoint,
        &barometric,
        &relative_light,
        &battery,
        &trailer))) {
        printf ("%s Scan returned %d\n",mymq_queue,res);
        return -1; // didn't get the right number of parameters
    }
    if (flag_D) {
        printf ("%s sscanf returned -> %d from %s\n",mymq_queue,res,input_buffer);
        printf ("%s values -> hdr:%c,t:%.1f,h:%d,d:%.1f,P:%.2f,l:%.1f,b:%.2f,trl:%c\n",
            mymq_queue,
            header,
            temperature,
            humidity,
            dewpoint,
            barometric,
            relative_light,
            battery,
            trailer);
    }
}

```

```

    if (flag_v)
        printf("%s Chk_msg was:%s\n",mymq_queue,input_buffer);
    }
    else {
        if (keepRunning) { // could have been a 'force exit' !
            printf("%s invalid message -> %s\n",mymq_queue,input_buffer);
        }
        return -1;
    }
    return 0;
}
//-----
void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n",(void*) &mosq,obj,result);
}
//-----
void message_callback(struct mosquitto *mosq, void *obj,
                      const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n",(void *) &mosq,
        (void *) &obj, (void *) &message);
}
//-----
void intHandler(int dummy) {
    printf("\n%s sigint produced -> %d\n",mymq_queue,dummy);
    printf("%s Marking System for close, please be patient \n",mymq_queue);
    keepRunning = 0;
}
//-----
void usage()
{
    printf("Berkeley Intel Environ_DT_MQ FTDI\n");
    printf("environ_dt_mq_FTDI [-h host] [-p port] [-d device] [-D] [-v] [-n queue name]\n\n");
    exit(0);
}
//-----
int is_number (char *s) {
    int res = 0;
    for (; s && *s; s++) {
        res = 1;
        if ((*s < '0') || (*s > '9'))
            return (0);
    }
    return (res);
}
//-----
void flags_init (int argc, char *argv[]) {
    int i;
    for (i=1; i<argc; i++) {
        if (! strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (! strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (! strcmp(argv[i], "-h") && (i+1 < argc)) {
            flag_h = 1; arg_host = argv[++i]; continue;
        }
        if (! strcmp(argv[i], "-d") && (i+1 < argc)) {
            flag_d = 1; arg_dev = argv[++i]; continue;
        }
        if (! strcmp(argv[i], "-n") && (i+1 < argc)) {
            flag_n = 1; arg_queue = argv[++i]; continue;
        }
        fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
        usage();
    }
    if (flag_d && arg_dev) {
        sprintf(serial_number,"%s", arg_dev);
    }
    if (flag_n && arg_queue) {
        sprintf(mymq_queue,"%s", arg_queue);
    }
    if (flag_p && arg_port) {
        port = atoi(arg_port);
        printf("port: = %d\n", port);
    }
    if (flag_h && arg_host) {
        struct hostent *hp;
        hp = gethostbyname(arg_host);
    }
}

```

```

    if (! hp) {
        fprintf(stderr, "unknown host \"%s\"\n", arg_host);
        exit(1);
    }
    //printf("host-a: %08x\n", host);
    host = 0;
    for (i=0; i<4; i++) {
        int octet = hp->h_addr[i] & 0xff;
        //printf("%d\n", octet);
        host = (host << 8) | octet;
    }
    //printf("host-b: %08x\n", host);
}
printf("Broker is --> %d.%d.%d.%d : %d\n",
      (host >> 24) & 0xff, (host >> 16) & 0xff,
      (host >> 8) & 0xff, (host >> 0) & 0xff, port);
printf("Queue is --> %s\n", mymq_queue);
printf("Device is --> %s\n", serial_number);
printf("Debug is --> %d\n", flag_D);
printf("Verbose is --> %d\n", flag_v);
fflush(NULL);
}
//-----

int main(int argc, char *argv[]) {
    int res, mos_res;
    int rc;
    char client_id[24];
    char mq_buf[MQ_BUF];
    char time_buf[SM_BUF];
    struct mosquito *mosq;

    res = check_devices();
    signal(SIGINT, intHandler); // assign signal handler
    signal(SIGTERM, intHandler);

    flags_init(argc, argv);

    mosquito_lib_init();
    memset(client_id, 0, 24);
    snprintf(client_id, 23, "environ_%d", getpid());
    mosq = mosquito_new(client_id, true, user_ptr);
    if (mosq) {
        mosquito_connect_callback_set(mosq, connect_callback);
        mosquito_message_callback_set(mosq, message_callback);
        rc = mosquito_connect(mosq, arg_host, port, 60);
        if (rc) {
            printf("%s Link Down %d\n", mymq_queue, rc);
            link_status = LINK_DOWN;
        }
        else {
            printf("%s Link UP\n", mymq_queue);
            link_status = LINK_UP;
        }
    }
}
// start network loop thread
mosquito_loop_start(mosq);

// main loop
res = 0;
while (keepRunning) {
    res = process_msg();
    if (!res) {
        res = tstring(time_buf, SM_BUF);
        snprintf(mq_buf, MQ_BUF, MQ_STR,
            mymq_queue,
            res?"1970-01-01 00:00:00.000":time_buf,
            temperature,
            humidity,
            dewpoint,
            barometric);
    }
}

```

```

switch (link_status) {
case LINK_UP:
    mos_res = mosquitto_publish(mosq, NULL, mymq_queue, strlen(mq_buf),mq_buf, 0,false);
    if (mos_res) { // not good !
        link_status = LINK_DOWN;
        printf ("%s Link DOWN \n",mymq_queue);
    }
    break;

case LINK_DOWN:
    mos_res = mosquitto_reconnect(mosq); // attempt reconnect if broker went down
    if (!mos_res) {
        link_status = LINK_UP;
        printf ("%s Link UP \n",mymq_queue);
    }
    break;

default:
    printf ("%s Internal error on link status\n",mymq_queue);
    break;

}
if (flag_D) {
    printf ("%s:%s\n",mymq_queue,mq_buf);
}
}
}
// printf ("%s Res returned -> %d\n",mymq_queue,res);
mosquitto_disconnect(mosq);
mosquitto_destroy(mosq);
printf ("%s Disconnected from broker\n",mymq_queue);
fflush(NULL);
return EXIT_SUCCESS;
}

```

Appendix K Software for Analog and Digital Acquisition for Monitoring of Plasma Systems

Code for driving a set of USB based Analog to Digital converters and Digital Inputs and Outputs.

```
// #includes here
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <pthread.h>

#include <sys/time.h>
#include <time.h>
#include <signal.h>
#include <semaphore.h>
#include <pthread.h>

#include <errno.h>
#include <netdb.h>

#include "wr3_time.h"
#include <mosquitto.h>

#include "iniparser.h"

// # defines etc

#define MQTT_PORT 1883
#define LOCALHOST 0x7f000001
#define LINK_DOWN 0
#define LINK_UP 1
#define KEEPALIVE 60

#define SM_BUF 256
#define MQ_BUF 4*256

#if 0
#define ADC0 0
#define ADC1 1
#define ADC2 2
#define ADC3 3
#define ADC4 4
#define ADC5 5
#define ADC6 6
#define ADC7 7
#define DI 8
#define RATE0 9
#define RATE1 10
#define RATE2 11
#define RATE3 12
#define RATE4 13
#define RATE5 14
#define RATE6 15
#define RATE7 16
#define RATE8 17
#define RATE9 18
#define RATE10 19
#define RATE11 20
#define COUNT 21
#define CLEAR 22
#endif
```

```

#define NO_ACHAN 8

#define RUNNING 2
#define STOPPED 1

// Sample rate per scan list element = 750,000 / srate
// So for example .. we have
#define SCAN_RATE_ELEMENTS 11
#define SRATE_MAX 65535
#define SRATE_MIN (75 * SCAN_RATE_ELEMENTS)

// We want ~100Hz
// #define SRATE 7500

// We want ~10Hz
// #define SRATE 75000

// Further reduce the datarate (only every 5th element at 10Hz => 2Hz !)
// #define REDUCTION 5
// OR
// could use SRATE to set 2Hz as per KB
//
// #define SRATE (750/5) /* 150 = 2Hz */

#if 0
#define SRATE 37500
#endif
#define REDUCTION 10

//-----

// Macros !

#if 0
#define DIE(A,fmt, ...) ;
#else
#define DIE(A, fmt, ...) \
    fprintf(stderr, (fmt), ##__VA_ARGS__); \
    fflush(NULL); \
    exit(A);
#endif

#if 0
#define vprintf(fmt, ...) ;
#else
#define vprintf(fmt, ...) \
    if (flag_v) { \
        printf((fmt), ##__VA_ARGS__); \
        fflush(NULL); \
    }
#endif

#if 1
#define VPRINT(A) do { if (flag_v) { printf A; } } while (0)
#define VDPRINT(A) do { if (flag_v && flag_D) { printf A; } } while (0)
#else
#define VPRINT(A)
#define VDPRINT(A)
#endif

//-----

// Inifile stuff
char inifile[SM_BUF]="ad0.ini";
char *arg_inifile = inifile;

// Mosquitto
char sbroker[SM_BUF] = "localhost";
unsigned int broker = 0x7f000001; // localhost IPV4 only
char clientid[SM_BUF];
int mqttport = MQTT_PORT;
char topicname[SM_BUF] = "ICE121/AT0";

```

```

char sensorname[SM_BUF] = "AT0";
int persist = false;
int keepalive = KEEPALIVE;
int link_status = LINK_DOWN;

// Device stuff
char devname[SM_BUF]; // no default
int fd_di149 = 0;      // file descriptor

int analog_input[NO_ACHAN];
int digital_input = 0;
int digital_output = 0;
int count_input = 0;
int rate_input = 0;
int rate_range = 0;
int srate = 0;          // what is the rate
char sn[SM_BUF];        // serial number storage space
int slist_len;          // keep a count of items to be scanned

// Some IO buffers
char input_buffer[MQ_BUF];
char output_buffer[MQ_BUF];

char *user_ptr = NULL;

// Acquisition stuff

#ifdef 0

char *slistrate[][2] = {
    { "x0009", "RATE0" },
    { "x0109", "RATE1" },
    { "x0209", "RATE2" },
    { "x0309", "RATE3" },
    { "x0409", "RATE4" },
    { "x0509", "RATE5" },
    { "x0609", "RATE6" },
    { "x0709", "RATE7" },
    { "x0809", "RATE8" },
    { "x0909", "RATE9" },
    { "x0a09", "RATE10" },
    { "x0b09", "RATE11" },
    { "x0b09", "RATEXX" }, // 12 illegal
    { "x0b09", "RATEXX" }, // 13 illegal
    { "x0b09", "RATEXX" }, // 14 illegal
    { "x0b09", "RATEXX" }, // 15 illegal
};

char *slist[][2] = {
/* A to D */
    { "x0000", "ADC0" },
    { "x0001", "ADC1" },
    { "x0002", "ADC2" },
    { "x0003", "ADC3" },
    { "x0004", "ADC4" },
    { "x0005", "ADC5" },
    { "x0006", "ADC6" },
    { "x0007", "ADC7" },
    { "x0008", "DI" },
    { "x0009", "ILLEGAL" },
    { "x000a", "COUNT" },
    { "x000b", "ILLEGAL" },
    { "x000c", "ILLEGAL" },
    { "x000d", "ILLEGAL" },
    { "x000e", "ILLEGAL" },
    { "xffff", "CLEAR" },
};

```



```

};

#endif

float scale[NO_ACHAN] = { 19.5E-3 / 4.0 , /* chan 0 = 4 counts per 19.5 mV*/
    19.5E-3 / 4.0 , /* chan 1 */
    19.5E-3 / 4.0 , /* chan 2 */
    19.5E-3 / 4.0 , /* chan 3 */
    19.5E-3 / 4.0 , /* chan 4 */
    19.5E-3 / 4.0 , /* chan 5 */
    19.5E-3 / 4.0 , /* chan 6 */
    19.5E-3 / 4.0 /* chan 7 */
};

volatile int keep_going = 1;
volatile int mode = STOPPED; // lock for DO

struct sigaction new_action, old_action;
sem_t data_lock;
pthread_t mq_worker;

// The data !
int di03_in; // 4 bits of Digital in
float adcf[NO_ACHAN]; // Floats updated by get_scan
int rate_in = 0; // The rate calculated by di149 and read in
int count_in = 0; // the count appearing on pin DI3
long long read_count = 0; // how many times we have polled since up

// usage flags
int flag_t = 0; // /* Test mode */
int flag_v = 0; // /* verbose */
int flag_D = 0; // /* debug */
int flag_i = 0; // /* inifile .. mandatory */

// constants

//-----

char tohex(int chr) {
    chr &= 0x0f;
    switch(chr) {
        case 0x00: return '0'; case 0x01: return '1';
        case 0x02: return '2'; case 0x03: return '3';
        case 0x04: return '4'; case 0x05: return '5';
        case 0x06: return '6'; case 0x07: return '7';
        case 0x08: return '8'; case 0x09: return '9';
        case 0x0a: return 'a'; case 0x0b: return 'b';
        case 0x0c: return 'c'; case 0x0d: return 'd';
        case 0x0e: return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}

//-----

void termination_handler (int signum) {
    keep_going = 0; // just mark that we are done !
    // sigaction (SIGINT, &old_action, NULL); // load up the old action
    if (signum > 10000) {
        printf("Signum is > 10K .. shud never happen !\n");
    }
}

//-----

long long current_timestamp(int RetSeconds) {
    int res;
    struct timespec ts;

```

```

res = clock_gettime(CLOCK_REALTIME, &ts);
if (res) {
    perror("Could not read clock");
    strerror(errno);
}
long long milliseconds = ts.tv_sec*1000LL +
    ts.tv_nsec/1000000; // calculate ms
// printf("Timestamp %sS ->%lld", (RetSeconds ? "" : "m"),
//     (RetSeconds ? (milliseconds/1000) : milliseconds));
return (RetSeconds ? (milliseconds/1000) : milliseconds);
}

//-----

int tstring(char *buf, int len) {
int res;
struct timespec ts;
struct tm t;

res = clock_gettime(CLOCK_REALTIME, &ts);
if (res) {
    perror("could not read clock");
    strerror(errno);
}

tzset();
if (localtime_r(&(ts.tv_sec), &t) == NULL) {
    return 1;
}
res = strftime(buf, len, "%F %T", &t);

if (0 == res) {
    return 2;
}
len -= res - 1; // get the formatted length

// nS reporting
// res = snprintf(&buf[strlen(buf)], len, "%.09ld", ts.tv_nsec);

// want mS reporting
res = snprintf(&buf[strlen(buf)], len, "%.03ld", (ts.tv_nsec/1000000));

if (res >= len) {
    return 3;
}

return 0;
}

//-----

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n", (void*) &mosq, obj, result);
}

//-----

void message_callback(struct mosquitto *mosq, void *obj,
    const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n", (void *) &mosq,
        (void *) &obj, (void *) &message);
}

//-----

int is_number (char *s) {
    int res = 0;
    for (; s && *s; s++) {
        res = 1;
        if ((*s < '0') || (*s > '9'))
            return (0);
    }
    return (res);
}

```

```
//-----

int set_interface_attribs (int fd, int speed, int parity) {
    struct termios tty;

    memset (&tty, 0, sizeof tty);
    if (tcgetattr (fd, &tty) != 0) {
        fprintf(stderr,"error %d from tcgetattr", errno);
        return -1;
    }

    cfsetospeed (&tty, speed);
    cfsetispeed (&tty, speed);

    tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;    // 8-bit chars
    // disable IGNBRK for mismatched speed tests; otherwise receive break
    // as \000 chars
    tty.c_iflag &= ~IGNBRK;                       // disable break processing
    tty.c_lflag = 0;                               // no signaling chars, no echo,
                                                // no canonical processing
    tty.c_oflag = 0;                               // no remapping, no delays
    tty.c_cc[VMIN] = 0;                           // read doesn't block
    tty.c_cc[VTIME] = 5;                          // 0.5 seconds read timeout

    tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff ctrl

    tty.c_cflag |= (CLOCAL | CREAD);              // ignore modem controls,
                                                // enable reading
    tty.c_cflag &= ~(PARENB | PARODD);            // shut off parity
    tty.c_cflag |= parity;
    tty.c_cflag &= ~CSTOPB;
    tty.c_cflag &= ~CRTSCTS;

    if (tcsetattr (fd, TCSANOW, &tty) != 0) {
        fprintf(stderr,"error %d from tcsetattr", errno);
        return -1;
    }
    return 0;
}

//-----

void set_blocking (int fd, int block_on_chars) {
    struct termios tty;

    memset (&tty, 0, sizeof tty);
    if (tcgetattr (fd, &tty) != 0) {
        fprintf(stderr,"error %d from tggetattr", errno);
        return;
    }

    // tty.c_cc[VMIN] = should_block ? 1 : 0;
    tty.c_cc[VMIN] = block_on_chars;
    tty.c_cc[VTIME] = 5;                          // 0.5 seconds read timeout

    if (tcsetattr (fd, TCSANOW, &tty) != 0)
        fprintf(stderr,"error %d setting term attributes", errno);
}

//-----

int DO(int fd, int value) {
    int n;
    char obuf[8];

    obuf[0]=0;
    obuf[1]='D';
    obuf[2]='0';
    obuf[3]=tohex(value); // '0' ... '9','a'...'f'
    obuf[4]=0;
    n = write(fd,obuf,4); // .. no echo !
    if (n==34) {
        return 0;
    }
}
```

```

    return -1;
}

//-----

int reset_count(int fd) {
int n;
char obuf[4];
obuf[0]=0;
obuf[1]='R';
obuf[2]='1';
obuf[2]=0;
n = write(fd,obuf,4); // .. no echo sometimes (ie bin in Scan!)
if (n==4) {
    return 0;
}
return -1;
}

//-----

int read_chars(int fd, int expected, char *buf) {
int n;

set_blocking (fd, expected);
memset(buf, 0, (expected+1));
n = read (fd, buf, expected);
if (n != expected) {
    return 1; // normal, comment for debug
    fprintf(stderr,
        "Read_char [err %d] expctd %d recvd %d chars, response is '%s'\n",
        errno,
        expected,
        n,
        buf);
    return 1;
}
return 0;
}

//-----

int write_read(int f, char *cmd, int wlen, char *buf, int rlen) {
int res;
char zero = 0;

write (f, &zero, 1); // clear the baffles
write (f, cmd, wlen); // send cmd
res = read_chars(f, rlen, buf); // read response
if (res) {
    return 1;
}
// printf ("\n[%s]\n",cmd);
return 0;
}

//-----

int set_scan_rate (int fd, int sr) {
int res;
char obuf[SM_BUF];
char ibuf[SM_BUF];

snprintf(obuf, SM_BUF, "srate %.5d\n", sr);
res = write_read(fd, obuf, 12, ibuf, 12); // sset srate
if (res) {
    fprintf(stderr,"Could not srate [%d]\n",sr);
    return 1;
}
return 0;
}

//-----

```

```

int set_scan_list(int fd, int sl) {
char obuf[SM_BUF];
char ibuf[SM_BUF];
int slot;
int res;
int slist_ptr = 0;

snprintf(obuf, SM_BUF, "slist %.2d %s\r", 0, "x0000"); // write ch 0 to clear
VDPRINT(("Set slist pos [%.2d] -> '%s' -> %s\n",
0,
"x0000",
"ADC0"));
res = write_read(fd, obuf, 15, ibuf, 15); // set scanlist 0 to ffff, clears it
if (res) {
return 1;
}

// fill from analog and then check rest
for (slot=0; slot < NO_ACHAN; slot++) {
if (analog_input[slot]) {
snprintf(obuf, SM_BUF, "slist %.2d x000%c\r", slist_ptr, tohex(slist_ptr));
VDPRINT(("Analog[%d] -> slist %.2d x000%c\n", slot, slist_ptr, tohex(slist_ptr)));
res = write_read(fd, obuf, 15, ibuf, 15); // set scanlist
if (res) {
return 1;
}
slist_ptr++;
}
else {
VDPRINT(("Analog[%d] not enabled\n", slot));
}
}

if (digital_input) {
snprintf(obuf, SM_BUF, "slist %.2d x000%c\r", slist_ptr, tohex(slist_ptr));
VDPRINT(("Digital Input -> slist %.2d x000%c\n", slist_ptr, tohex(slist_ptr)));
res = write_read(fd, obuf, 15, ibuf, 15); // set scanlist
if (res) {
return 1;
}
slist_ptr++;
}
else {
VDPRINT(("Digital Input not enabled in slot %d\n", slist_ptr));
}
if (slist_ptr != sl) {
fprintf(stderr, "Slots count did not match %d:%d\n", slist_ptr, sl);
return -1;
}
VDPRINT(("Expect %d bytes with scanning of %d slots\n", (sl<=2), sl));

// TODO add rate and count stuff

return 0;
}

//-----

int open_di149(char *portname, int *f) {

printf("Attempting to open '%s'\n", portname);

*f = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
if (*f < 0) {
fprintf(stderr, "error %d opening %s: %s",
errno,
portname,
strerror (errno));
return -1;
}

set_interface_attribs (*f, B115200, 0); // set speed to 115,200 bps, 8n1, NP
set_blocking(*f, 0); // initially no blocking !!

```

```

    return 0;
}

//-----

int close_di149(int *f) {
int res;
res = write (*f, "\r\n\r\n", 4);    // No reply !!
if (res != 4) {
    fprintf(stderr,"Could not sync <cr> to di149\n");
}
res = write (*f, "stop\r", 5);      // stop any incoming
if (res != 5) {
    fprintf(stderr,"Could not send 'stop' to di149\n");
}
close (*f);
*f = 0;
return 0;
}

//-----
//
// If MIN = 0 and TIME > 0, TIME serves as a timeout value.
// The read will be satisfied if a single character is read, or
// TIME is exceeded (t = TIME *0.1 s). If TIME is exceeded, no
// character will be returned.
int drain (int fd) {
int n;
#define DC 8
int draincount = DC;
char ibuf[SM_BUF];
set_blocking(fd,0); // no blocking !! VMIN is 5 => 1/2 second
while (draincount) {
    printf("Draining %d\n",draincount);
    n = read (fd, ibuf, SM_BUF);
    // ~12K chars per second => 12 every ms =>
    usleep(100);
    draincount--;
    if (n < 0) continue;
    if (n > 0) {
        printf("Cleared %d from di-149\n",n);
    }
}
if (!draincount) {
    fprintf(stderr,"Timed out draining -> stopped %d\n",DC);
}
else {
    printf("Drained %d\n",draincount);
}
return 0;
}
//-----

int reset_di149(int fd) {
int res;
int i;
int n;
char ibuf[SM_BUF];

// sync seems to be <CR> .. lets send a few
res = write (fd, "\r\n\r\n", 4);    // No reply !!
if (res != 4) {
    DIE(1,"Could not write to device <cr> for sync\n");
}
res = write (fd, "stop\r", 5);      // stop any incoming, ignore error
usleep(50000); // 50 mS
res = write (fd, "stop\r", 5);
usleep(50000); // 50 mS
res = write (fd, "stop\r", 5);
usleep(50000); // 50 mS

vprintf("Attempting to purge buffer\n");
res = drain(fd);
i = 0;
do {

```

```

n = read(fd, ibuf, 1); // Purge the incoming buffer ! shud timeout
i++;
if (i > 128) {
    DIE(1,"Would not come ready, was still babbling\n");
}
} while (n > 0);
vprintf("Gone quiet after %d\n",i);

// Now incoming should be empty .. and he should be listening !

res = write_read(fd,"A0", 2, ibuf, 2+5); // A0 seems to work, gives 'DATAQ'
if (res) return 1;
res = write_read(fd,"asc\r", 4, ibuf, 4); // use ascii initially
if (res) return 1;

res = write_read(fd,"A0", 2, ibuf, 2+5); // check that it is a dataq
if (res) return 1;
vprintf ("Device is -> %c%c%c%c%c\n",ibuf[2],ibuf[3],ibuf[4],ibuf[5],ibuf[6]);

res = write_read(fd,"A1", 2, ibuf, 2+4); // check that it is a '1490'
if (res) return 1;
vprintf ("Type is -> %c%c%c%c%c\n",ibuf[2],ibuf[3],ibuf[4],ibuf[5]);

res = write_read(fd,"A2", 2, ibuf, 2+2); // check FW version
if (res) return 1;
vprintf ("FW Vers is -> %c.%c\n",ibuf[2],ibuf[3]);

res = write_read(fd,"A6", 2, ibuf, 2+10); // check Serial Number
if (res) return 1;
memcpy(sn, &ibuf[2],8); // only leftmost 8 hex digits are valid !
sn[8] = 0;
vprintf ("Serial Number is -> %s\n",sn);
return 0;
}

//-----
// This reads till it finds a sync .. and which point .. it
// reads one more 'block' leaving the 'next' read by the
// caller above will be aligned !

int find_sync(int fd, int slen) {
int i;
int res;
char ibuf[SM_BUF];

// find sync
ibuf[0] = 0x01;
i = 0;
vprintf("Find sync\n");
while (ibuf[0] & 0x01) { // look for the one SYNC char
res = read_chars(fd, 1, ibuf);
if (!res) {
vprintf("S%d=[%.2x]\n",i++,0xff & ibuf[0]);
if (i > (4 * slen)) { // allow for two blocks !
DIE(1,"We have looked too far for SYNC\n");
}
}
else {
DIE(1,"Error trying to SYNC : %d \n",res);
}
}
vprintf("Found sync, dump %d\n",(2*slen)-1);
res = read_chars(fd, (2*slen)-1, ibuf);
if (res) {
fprintf(stderr,"Did not get expected Chars on sync\n");
return 1;
}
vprintf("Next read should be aligned\n");
return 0;
}

//-----

int start(int fd) {
char ibuf[SM_BUF];

```

```

int res;

res = write_read(fd,"bin\r", 4, ibuf, 4); // make it binary
if (res) {
    fprintf(stderr,"Could not make it bin for start\n");
    return 1;
}
res = write_read(fd,"start\r", 6, ibuf, 6); // start it
if (res) {
    fprintf(stderr,"Could not start it\n");
    return 1;
}
mode = RUNNING;
return 0;
}

//-----

int stop(int fd) {
char ibuf[SM_BUF];
int res;

res = write_read(fd,"stop\r", 5, ibuf, 5); // stop it
if (res) {
    fprintf(stderr,"Could not stop it\n");
    return 1;
}
res = write_read(fd,"asc\r", 4, ibuf, 4); // make it ascii
if (res) {
    fprintf(stderr,"Could not make it ascii\n");
    return 1;
}
mode = STOPPED;
return 0;
}

//-----

int get_scan_elements(int fd, float *chan8, int *di, int *ri, int *ci) {
int i;
int slist_ptr = 0;
int adc[NO_ACHAN];
int res;
char ibuf[SM_BUF];

res = read_chars(fd, 2 * slist_len, ibuf);
if (ibuf[0] & 0x01) {
    printf("Lost sync -1:{%.2x} : 0:{%.2x} : +1:{%.2x}\n",
        0xff & ibuf[(slist_len*2) - 1],
        0xff & ibuf[0],
        0xff & ibuf[1]);

    return -1;
}
if (res) {
    fprintf(stderr,"Scan .. did not get expected characters\n");
    return -1;
}
// Assemble the various values
// ADC (8 channels)
for (i = 0; i < NO_ACHAN; i++) {
    if (analog_input[i]) {
        adc[i] = (ibuf[slist_ptr*2] >> 3) & 0x1f; //A[4:0] D[1:0] 0/1
        adc[i] |= ((ibuf[(slist_ptr*2)+1] & 0xfe) << 4); // A[11:5] 1

        adc[i] = adc[i] ^ 0x800; // invert most significant
        adc[i] |= adc[i] & 0x800 ? 0xffff800 : 0; // sign extend to int
        chan8[i] = adc[i] * scale[i];
        slist_ptr++;
    }
}
// Digital Input
if (digital_input) {
    *di = ((ibuf[slist_ptr] & 0x6) >> 1) | // assemble from the bits D0 D1
        (ibuf[slist_ptr+1] & 0xc); // D2 D3
}

```



```

    }

#ifdef 0
// Rate Input
*ri = ((ibuf[18] & 0xfe) >> 1) | // assemble B7-B1 -> D6-D0
      ((ibuf[19] & 0xfe) >> 2); // and B7-B1 -> D13-D7 => 14 bits rate
// Count Input
*ci = ((ibuf[20] & 0xfe) >> 1) | // assemble B7-B1 -> D6-D0
      ((ibuf[21] & 0xfe) >> 2); // and B7-B1 -> D13-D7 => 14 bits count
#endif

    return 0;
}

//-----

void init_var(void) {
int i;
    read_count = 0;
    count_in = 0;
    rate_in = 0;
    di03_in = 0;
    for (i=0; i<NO_ACHAN; i++) {
        adcf[i] = 0.0;
    }
}

//-----

static void * post_thread(void *arg) {
int decimate = 0;

char mq_buf[MQ_BUF];
char time_buf[SM_BUF];
int res;
char client_id[24];
struct mosquitto *mosq;

    vprintf(">>Entered Worker thread with argp -> %p\n",arg);

    mosquitto_lib_init();
    memset(client_id, 0, 24);
    snprintf(client_id, 23, "di149_%d", getpid());
    mosq = mosquitto_new(client_id, true, user_ptr);
    if (mosq) {
        mosquitto_connect_callback_set(mosq, connect_callback);
        mosquitto_message_callback_set(mosq, message_callback);
        res = mosquitto_connect(mosq, sbroker, mqport, keepalive);
        if (res) {
            printf("%s Link Down %d\n",sensorname,res);
            link_status = LINK_DOWN;
        }
        else {
            printf("%s Link UP\n",sensorname);
            link_status = LINK_UP;
        }
    }
// start network loop thread
mosquitto_loop_start(mosq);

while (keep_going) {
    sem_wait(&data_lock);    // down semaphore

    if ((decimate/REDUCTION) * REDUCTION == decimate) { // just byte reduction !
        if (keep_going) { // remove that last one !

#define Q_DT_FMT "'%s', '%s',0x%.16x," \
/* 0 1 2 3 4 5 6 7 */ \
        "%.4f,%.4f,%.4f,%.4f,%.4f,%.4f,%.4f,%.4f" // 8 floats

            res = tstring(time_buf, SM_BUF); // get string time-date
            snprintf (mq_buf, MQ_BUF, Q_DT_FMT,
                sensorname,

```

```

        res?"1970-01-01 00:00:00.000":time_buf,
        di03_in,
        adcf[0], // 0
        adcf[1], // 1
        adcf[2], // 2
        adcf[3], // 3
        adcf[4], // 4
        adcf[5], // 5
        adcf[6], // 6
        adcf[7] // 7
    );

    // printf("Count = %d\n",count_in);
    // printf("Rate = %d\n",rate_in);

    switch (link_status) {
    case LINK_UP:
        res = mosquitto_publish(mosq,
                                NULL,
                                topicname,
                                strlen(mq_buf),
                                mq_buf,
                                0,
                                false);
        if (res) { // not good !
            link_status = LINK_DOWN;
            printf ("%s -> link down \n",topicname);
        }
        break;

    case LINK_DOWN:
        res = mosquitto_reconnect(mosq); // broker down, attempt reconnect
        if (!res) {
            link_status = LINK_UP;
            printf ("%s -> link up \n",topicname);
        }
        break;

    default:
        printf ("%s Internal error on link status\n",topicname);
        break;

    } // link_status ?

    if (flag_D) {
        printf ("%s:%s\n",sensorname,mq_buf);
    }
    } // keep_going one last time
    } // decimate
    } // while keep_going

    mosquitto_disconnect(mosq);
    mosquitto_destroy(mosq);
    vprintf ("%s Disconnected from broker\n",topicname);

    return NULL;
}

//-----

void usage() {
    printf("Berkeley Intel DI149 \n");
    printf("di149 -c configfile.ini [-D] [-v] [-t]\n");
    printf(" where:\n");
    printf(" -c config.ini is mandatory config file\n");
    printf(" -D debug -> dumps everything\n");
    printf(" -v verbose -> gives status as processing is being undertaken\n");
    printf("\n");

    printf(" Primary Message Queue Format is:- \n"
           "tAAA, (decimal 0.00 to 25.00)\n"
           "tBBB (decimal -1.00 to +1.00), [integer 0:1]\n");
    printf("\n");
}

```

```

    exit(0);
}

//-----

// Calc checksum to insert in 'three digit [000 to 255] checksum and ':'
int calc_tail(char * mq_bf, int mlen) {
    int i;
    int sum = 0;

    for (i=0; i<mlen-4; i++) { // exclude already add 'digits' and ':'
        sum += mq_bf[i];
    }
    sum &= 0xff;
    sprintf(&mq_bf[mlen-4], "%03d:", sum); // left justified 3 digits decimal

    return 0;
}

//-----

int build_shaper_mq_header(char *buf,
                           int buf_len,
                           char *sensor_name,
                           char *data_type,
                           char *data_structure,
                           int elements,
                           char *separator,
                           int msg_len) {

    int res;
    char tim_buf[SM_BUF];

    res = wr3_date(tim_buf, SM_BUF, DEC_4_PLACE);
    if (res) {
        fprintf(stderr, "Error in datetime function\n");
    }

    res = snprintf(buf, buf_len, "%s;%s;%s;%s;%d;%s;%d",
                  sensor_name,
                  tim_buf,
                  data_type,
                  data_structure,
                  elements,
                  separator,
                  msg_len);

    if (buf_len < res) {
        fprintf(stderr, "Could not pack header, buf too small %d:%d\n", buf_len, res);
        return -1;
    }
    return res;
}

//-----

int load_ini_parms(char * ini_file) {
    char anunit[SM_BUF];
    char client_id[24]; // temp
    const char *cp;
    dictionary *d = NULL;
    struct hostent *hp;
    int octet;
    int host;
    int i,j;

    d = iniparser_load(ini_file);
    if (NULL == d) {
        fprintf(stderr, "Could not load %s INI file !!\n", ini_file);

```

```

    exit(1);
}

// Adio stuff

for (i = 0; i<NO_ACHAN; i++) {
    snprintf(anunit,SM_BUF,"%s:Analog%d","ADIO",i);
    VDPRINT(("Read --> %s\n",anunit));
    analog_input[i] = iniparser_getint(d, anunit, 0);
}

snprintf(anunit,SM_BUF,"%s:%s","ADIO","DI");
VDPRINT(("Read --> %s\n",anunit));
digital_input = iniparser_getint(d, anunit, 0);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","DO");
VDPRINT(("Read --> %s\n",anunit));
digital_output = iniparser_getint(d, anunit, 0);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","CI");
VDPRINT(("Read --> %s\n",anunit));
count_input = iniparser_getint(d, anunit, 0);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","RI");
VDPRINT(("Read --> %s\n",anunit));
rate_input = iniparser_getint(d, anunit, 0);

slist_len = 0;
for (i = 0; i<NO_ACHAN; i++) {
    if (analog_input[i]) {
        slist_len++;
    }
}
if (digital_input) {
    slist_len++;
}
}
#if 0
// reserved for future use
if (count_input) {
    slist_len++;
}
if (rate_input) {
    slist_len++;
}
}
#endif

// must be at least One thing to scan !
if (slist_len) {
    VDPRINT(("Scanning list has %d elements\n",slist_len));
}
else {
    fprintf(stderr,"Forcing scan of Digital Input\n");
    digital_input = 1; // have to scan something
}

snprintf(anunit,SM_BUF,"%s:%s","ADIO","RI_RANGE");
VDPRINT(("Read --> %s\n",anunit));
rate_range = iniparser_getint(d, anunit, 0);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","srate");
VDPRINT(("Read --> %s\n",anunit));
srate = iniparser_getint(d, anunit, 0);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","device");
VDPRINT(("Read --> %s\n",anunit));
cp = iniparser_getstring(d, anunit, "NoDevice");
snprintf(devname, SM_BUF, "%s", cp);

snprintf(anunit,SM_BUF,"%s:%s","ADIO","sensorname");
VDPRINT(("Read --> %s\n",anunit));
cp = iniparser_getstring(d, anunit, "AD0");
snprintf(sensorname, SM_BUF, "%s", cp);

```

```

// Mosquitto Stuff

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "topicname");
VDPRINT(("Read --> %s\n", anunit));
cp = iniparser_getstring(d, anunit, "AD0");
snprintf(topicname, SM_BUF, "%s", cp);

// set up a reasonable random client id (can pass NULL to procedure!)
memset(client_id, 0, 24); // generate a temporary one using PID
snprintf(client_id, 23, "adio_%d", getpid());

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "clientid");
VDPRINT(("Read --> %s\n", anunit));
cp = iniparser_getstring(d, anunit, client_id);
snprintf(clientid, 23, "%s", cp); // only 24 chars for client id (inc NULL)

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "broker");
VDPRINT(("Read --> %s\n", anunit));
cp = iniparser_getstring(d, anunit, "localhost");
snprintf(sbroker, SM_BUF, "%s", cp);
// check the broker, and see if addressable
hp = gethostbyname(cp);
if (! hp) {
    fprintf(stderr, "Unknown host \"%s\"\n", cp);
    exit(1);
}
host = 0;
for (j=0; j<4; j++) {
    octet = hp->h_addr[j] & 0xff;
    //printf("%d\n", octet);
    host = (host << 8) | octet;
}
broker = host;
// VPRINT("host: %08x\n", host);

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "mqport");
VDPRINT(("Read --> %s\n", anunit));
mqport = iniparser_getint(d, anunit, MQTT_PORT);

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "keepalive");
VDPRINT(("Read --> %s\n", anunit));
keepalive = iniparser_getint(d, anunit, KEEPALIVE);

snprintf(anunit, SM_BUF, "%s:%s", "MOSQUITTO", "persist");
VDPRINT(("Read --> %s\n", anunit));
persist = iniparser_getboolean(d, anunit, false);

iniparser_freedict(d);
d = NULL;

return 0;
}

//-----

void flags_init (int argc, char *argv[]) {
int i;
for (i=1; i<argc; i++) {
    if (! strcmp(argv[i], "-v")) { flag_v = 1; continue; }
    if (! strcmp(argv[i], "-t")) { flag_t = 1; continue; }
    if (! strcmp(argv[i], "-D")) { flag_D = 1; continue; }
    if (! strcmp(argv[i], "-c") && (i+1 < argc)) {
        flag_i = 1; arg_inifile = argv[i+1]; continue;
    }

    fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
    usage();
}
}

```

```

if (flag_i && arg_inifile) {
    if (strlen(arg_inifile) < sizeof(inifile)) {
        sprintf(inifile,"%s", arg_inifile);
    }
}

if (!flag_i) {
    fprintf(stderr, "You must specify an INI file\n");
    usage(); // does not return
}

load_ini_parms(inifile); // read some ini parms

printf("Inifile is      --> %s\n", inifile);
printf("Broker is      --> %d.%d.%d.%d : %d\n\r",
        (broker >> 24) & 0xff,
        (broker >> 16) & 0xff,
        (broker >> 8) & 0xff,
        (broker >> 0) & 0xff, mqport);
printf("ClientID is    --> %s\n", clientid);
printf("Topic is       --> %s\n", topicname);
printf("Sensor is      --> %s\n", sensorname);
printf("Device is      --> %s\n", devname);
printf("Debug is       --> %d\n", flag_D);
printf("Verbose is     --> %d\n", flag_v);
printf("Test Mode is   --> %d\n", flag_t);
for (i=0; i<NO_ACHAN; i++) {
    if (analog_input[i]) {
        printf("Analog_channel[%d] --> %d\n", i, analog_input[i]);
    }
}
printf("Digital Input   --> %d\n", digital_input);
printf("Srate is        --> %d\n", srate);

fflush(NULL);
}
//-----

int main(int argc, char *argv[]) {
    int res;
    long long time;

    // Set up the structure to specify the new action.
    sigemptyset (&new_action.sa_mask);
    new_action.sa_handler = termination_handler;
    new_action.sa_flags = 0;
    sigaction (SIGINT, NULL, &old_action);
    sigaction (SIGINT, &new_action, NULL);

    flags_init(argc, argv);
    init_var();
    time = current_timestamp(0);
    vprintf("Time starting is %lld\n",time)

    // Scan everything ! (8 analog chanel, 4 bit DI, Rate, Count )

    res = open_di149(devname, &fd_di149);
    if (res) {
        DIE(1,"Failed to open di149 [%s]\n",devname);
    }
    res = reset_di149(fd_di149);
    if (res) {
        DIE(1,"Failed to initialize di149\n");
    }

    res = set_scan_list(fd_di149, slist_len);
    if (res) {
        DIE(1,"Could not set slist\n");
    }
    res = set_scan_rate(fd_di149, srate);
    if (res) {
        DIE(1,"Could not set SRATE : %d\n",srate);
    }
    vprintf("Di149 now up and initialized\n");
}

```

```

res = start(fd_di149);
if (res) {
    goto done;
}
res = find_sync(fd_di149,slist_len); // This leaves 'Next' read to be aligned
if (res) {
    DIE(1,"Could not find sync\n");
}

sem_init(&data_lock, 0, 0); /* initialize semaphore to 0, not ready yet */
/* second param = 0 - semaphore is local */

res = pthread_create(&mq_worker, NULL, &post_thread, NULL);
if (res) {
    fprintf(stderr,"Could not create thread !\n");
    goto done;
}

while (keep_going) {
// main loop
// expect 2 bytes per enabled channel ... since we have
// 'scan_elements' (8 adc + 1 DI(packed))
// we expect 2 * SCAN_RATE_ELEMENTS chars (eg 22)

res = get_scan_elements(fd_di149, &adcf[0], &di03_in, &rate_in, &count_in);
if (res) {
    find_sync(fd_di149,slist_len);
}

read_count ++;
if (((read_count / REDUCTION) * REDUCTION) == read_count) { // decimate ?
    // Let the worker thread post !
    sem_post(&data_lock); /* up semaphore */
} // want to decimate the data_rate !
} // end while keep-going

// accept_input() and possibly write output
//

done:

keep_going = 0; // force for debug, if we are jumping here
res = stop(fd_di149);
if (res) {
    fprintf(stderr,"Could not stop di149, but will continue\n");
}

vprintf("No of records read was : %lld\n",read_count);
time = current_timestamp(0) - time;
printf("Delta time : ending is %lld mS\n",time);
sem_post(&data_lock); // send one to free it !
vprintf("Waiting for thread to join !\n");
pthread_join(mq_worker, NULL);
vprintf("Delete the semaphore !\n");
sem_destroy(&data_lock); /* destroy semaphore */
drain(fd_di149);
close_di149(&fd_di149);
return EXIT_SUCCESS;
}

```

Appendix L Software for Monitoring the Optical Spectrum of Plasma Systems operating on Linux®

This code drives the Ocean Optics USB2K and HR4000 range of Optical Spectrum Analyzers for monitoring the optical emission of the plasma etcher systems.

```
// #includes here
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <pthread.h>
#include <signal.h>
#include <sys/time.h>
#include <errno.h>
#include <netdb.h>
#include "usb2k.h"

#include <mosquitto.h>

#include <Wrapper.h>
#include <ArrayTypes.h>
#include <ContinuousStrobeImpl.h>
#include <JString.h>

#include "wr3_time.h"

#include "iniparser.h"

// # defines etc
#define SM_BUF 256
#define LG_BUF 1024
#define LOCALHOST 0x7f000001
#define MQTT_PORT 1883
#define KEEPALIVE 60

#define LINK_DOWN 0
#define LINK_UP 1

#define MAX_SPEC 4
#define MAX_PIXELS 4096
#define MIN_PIXELS 256

#define SHUTTER_MIN 5000; /* 5mS */
#define SHUTTER_MAX 100000 /* useconds */
#define SHUTTER_DELTA 5000; /* useconds */
#define MAX_PIXEL_VALUE 14745.0 /* 90 percent of 16384 */

#define MYSLEEP 75000 /* 75 milliseconds -> 10 samples per second */

#if 0
#define likely(x)    __builtin_expect((x),1)
#define unlikely(x)  __builtin_expect((x),0)
#else
// and the updated ones .. (!! -> cast long to bool)
#define likely(cond)  (__builtin_expect (!! (cond), 1))
#define unlikely(cond) (__builtin_expect (!! (cond), 0))
#endif

#if 1
#define VPRINT(A) do { if (flag_v) { printf A; } } while (0)
#define VDPRINT(A) do { if (flag_v && flag_D) { printf A; } } while (0)
#else
#define VPRINT(A)
#endif
```



```

#define VDPRINT(A)
#endif

#define STEP(A) printf(A "->"); getchar(); printf("\n"); /* use as STEP("1"); */

// globals

char inifile[LG_BUF]="spectrometer.ini";
char *arg_inifile = inifile;

volatile int keepRunning = 1;
WRAPPER_T wrapperHandle;

struct spec_object {
    int this_unit;
    volatile int parameterChange;
    volatile int integrationTime;
    volatile int boxCarWidth;
    volatile int scansAve;
    volatile int electricalDark;
    volatile double pixel_threshold;
    char queueName[LG_BUF];
    char sensorName[LG_BUF];
    char clientid[25]; /* 24 bytes, allow for single terminator for strings */
    char sbroker[LG_BUF];
    int broker;
    int port;
    int keepalive;
    char uname[SM_BUF];
    char pword[SM_BUF];
    int flag_e;
    int link_status;
    char *mosq_ptr;
    int more;
};

// usage flags
int flag_v = 0; /* verbose */
int flag_i = 0; /* ini file name */
int flag_D = 0; /* debug */

// constants

// externs

//-----
#define LO 0
#define HI 1

char tohex(unsigned char chr, int hilo) {
    if (hilo==HI)
        chr = (chr>>4) & 0x0f;
    else
        chr = chr & 0x0f;
    switch(chr) {
        case 0x00: return '0'; case 0x01: return '1';
        case 0x02: return '2'; case 0x03: return '3';
        case 0x04: return '4'; case 0x05: return '5';
        case 0x06: return '6'; case 0x07: return '7';
        case 0x08: return '8'; case 0x09: return '9';
        case 0x0a: return 'a'; case 0x0b: return 'b';
        case 0x0c: return 'c'; case 0x0d: return 'd';
        case 0x0e: return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}
//-----

```

```

#if 0
long long current_timestamp() {
    struct timeval te;
    gettimeofday(&te, NULL); // get current time (deprecated..)
    long long milliseconds = te.tv_sec*1000LL + te.tv_usec/1000; // calculate ms
    // printf("milliseconds: %lld\n", milliseconds);
    return milliseconds;
}
#else
long long current_timestamp() {
    int res;
    struct timespec ts;
    res = clock_gettime(CLOCK_REALTIME, &ts); {
        if (res)
            perror("could not read clock");
            strerror(errno);
        }
        long long milliseconds = ts.tv_sec*1000LL + ts.tv_nsec/1000000; // calc ms
        return milliseconds;
    }
#endif

//-----

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n", (void*) &mosq, obj, result);
}

//-----

void message_callback(struct mosquitto *mosq, void *obj, const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n", (void *) &mosq, (void *) &obj, (void *) &message);
}

//-----

void intHandler(int signo) {
    if (signo == SIGINT) {
        printf("Received SIGINT\n");
    }
    if (signo == SIGTERM) {
        printf("Received SIGTERM\n");
    }
    printf("Marking System for close, please be patient \n");
    keepRunning = 0;
}

//-----

void usage() {
    printf("Berkeley Intel Ocean-Optics usbOO_mq \n");
    printf("Licensed Software, Berkeley copyright\n");
    printf("usbOO_mq [-h host] [-D] [-v] [-c config ini file]\n");
    printf("\t -h [ip address numeric xx.xx.xx.xx] or [ip address DNS]\n");
    printf("\t -D low level debug\n");
    printf("\t -c config file (default is 'spectrometer.ini')\n");
    printf("\t -v verbose mode, (in conjunction with -D)\n");
    printf("\t -? print this message\n");
    exit(0);
}

//-----

int is_number (char *s) {
    int res = 0;
    for (; s && *s; s++) {
        res = 1;
        if ((*s < '0') || (*s > '9'))
            return (0);
    }
    return (res);
}

//-----

```

```

void print_spec(struct spec_object *s) {

    printf("Spectrometer --> %d\n", s->this_unit);
    printf("Broker is --> %s : %d\n\n",s->sbroker,s->port);
    printf("Broker is --> %d.%d.%d.%d : %d\n\n",
        ((s->broker) >> 24 & 0xff),
        ((s->broker) >> 16 & 0xff),
        ((s->broker) >> 8 & 0xff),
        ((s->broker) >> 0 & 0xff),
        s->port);
    printf("Sensor Name is --> %s\n", s->sensormname);
    printf("Queue is --> %s\n", s->queue_name);
    printf("Client ID is --> %s\n", s->clientid);
    printf("Keep Alive is --> %d\n", s->keepalive);
    printf("BoxCarWidth --> %d\n", s->boxCarWidth);
    printf("IntegrationTime --> %d\n", s->integrationTime);
    printf("ScansAve --> %d\n", s->scansAve);
    printf("Electrical Dark --> %d\n", s->electricalDark);
    printf("Pixel Threshold --> %.2f\n", s->pixel_threshold);

}

//-----

int load_ini_parms(char * ini_file, struct spec_object s[], int n) {
    int i,j;
    char sunit[SM_BUF];
    const char *cp;
    dictionary *d = NULL;
    struct hostent *hp;
    int octet;
    int host;

    d = iniparser_load(ini_file);
    if (NULL == d) {
        fprintf(stderr,"Could not load %s INI file !!\n",ini_file);
        exit(1);
    }

    for (i=0; i<n; i++) {
        snprintf(sunit,SM_BUF,"%s%d", "SPECTROMETER",i);
        VDPRINT(("+++++\n"));
        VPRINT(("Read '%s' for [%s]\n",ini_file,sunit));

        // Read the various parameters
        s[i].this_unit = i;

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"integrationtime");
        VDPRINT(("Read --> %s\n",sunit));
        s[i].integrationTime = iniparser_getint(d, sunit, 85000);

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"boxcarwidth");
        VDPRINT(("Read --> %s\n",sunit));
        s[i].boxCarWidth = iniparser_getint(d, sunit, 0);

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"scanperavg");
        VDPRINT(("Read --> %s\n",sunit));
        s[i].scansAve = iniparser_getint(d, sunit, 5);

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"electricaldark");
        VDPRINT(("Read --> %s\n",sunit));
        s[i].electricalDark = iniparser_getint(d, sunit, 10);

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"pixel_threshold");
        VDPRINT(("Read --> %s\n",sunit));
        s[i].pixel_threshold = iniparser_getdouble(d, sunit, 0.0);

        snprintf(sunit,SM_BUF,"%s%d:%s", "SPECTROMETER",i,"broker");
        VDPRINT(("Read --> %s\n",sunit));
        cp = iniparser_getstring(d, sunit, "localhost");
        snprintf(s[i].sbroker, LG_BUF, "%s", cp);

        hp = gethostbyname(cp);
        if (! hp) {
            fprintf(stderr, "Unknown host \"%s\"\n", cp);

```

```

    exit(1);
}
host = 0;
for (j=0; j<4; j++) {
    octet = hp->h_addr[j] & 0xff;
    //printf("%d\n", octet);
    host = (host << 8) | octet;
}
s[i].broker = host;
// VPRINT("host: %08x\n", host);

snprintf(sunit, SM_BUF, "%s%d:%s", "SPECTROMETER", i, "port");
VDPRINT(("Read --> %s\n", sunit));
s[i].port = iniparser_getint(d, sunit, MQTT_PORT);

snprintf(sunit, SM_BUF, "%s%d:%s", "SPECTROMETER", i, "keepalive");
VDPRINT(("Read --> %s\n", sunit));
s[i].keepalive = iniparser_getint(d, sunit, KEEPALIVE);

snprintf(sunit, SM_BUF, "%s%d:%s", "SPECTROMETER", i, "clientid");
VDPRINT(("Read --> %s\n", sunit));
cp = iniparser_getstring(d, sunit, "oops");
memset(&s[i].clientid[0], 0, 25); // set terminator
snprintf(s[i].clientid, 24, "%s", cp);

snprintf(sunit, SM_BUF, "%s%d:%s", "SPECTROMETER", i, "queueName");
VDPRINT(("Read --> %s\n", sunit));
cp = iniparser_getstring(d, sunit, "SP");
snprintf(s[i].queueName, LG_BUF, "%s", cp);

snprintf(sunit, SM_BUF, "%s%d:%s", "SPECTROMETER", i, "sensorName");
VDPRINT(("Read --> %s\n", sunit));
cp = iniparser_getstring(d, sunit, "OceanOptics");
snprintf(s[i].sensorName, LG_BUF, "%s", cp);

}

for (i=0; i< MAX_SPEC; i++) {
    printf ("---\n");
    print_spec(&s[i]);
}

iniparser_freedict(d);
d = NULL;

// check for 'matching clientids' .. this would corrupt queues
// as we would publish on top of existing
for (i=0; i<n; i++) {
    for (j=i+1; j<n; j++) {
        VPRINT(("Comparing clientid [%d][%d] %s' %s'\n", i, j, s[i].clientid, s[j].clientid));
        if (!strcmp(s[i].clientid, s[j].clientid)) {
            fprintf(stderr, "Matching Client IDs %d:%d: %s'\n", i, j, s[i].clientid);
            exit(2);
        }
        VPRINT(("Comparing queueName [%d][%d] %s' %s'\n", i, j, s[i].clientid, s[j].clientid));
        if (!strcmp(s[i].queueName, s[j].queueName)) {
            fprintf(stderr, "Matching Queue Names %d:%d: %s'\n", i, j, s[i].queueName);
            exit(3);
        }
    }
}

return 0;
}

//-----

// Calc checksum to insert in 'three digit [000 to 255] checksum and ':'
int calc_tail(char * mq_bf, int mlen) {
    int i;
    int sum = 0;

    for (i=0; i<mlen-4; i++) { // exclude 'digits' and ':'

```

```

    sum += mq_bf[i];
}
sum &= 0xff;
sprintf(&mq_bf[mflen-4], "%03d:", sum);

return 0;
}

//-----

int build_shaper_mq_header(char *buf,
                           int buf_len,
                           char *sensor_name,
                           char *data_type,
                           char *data_structure,
                           int elements,
                           char *separator,
                           int msg_len) {

int res;
char tim_buf[SM_BUF];

res = wr3_date(tim_buf, SM_BUF, DEC_4_PLACE);
if (res) {
    fprintf(stderr, "Error in datetime function\n");
}

res = snprintf(buf, buf_len, "%s;%s;%s;%s;%d;%s;%d",
               sensor_name,
               tim_buf,
               data_type,
               data_structure,
               elements,
               separator,
               msg_len);

if (buf_len < res) {
    fprintf(stderr, "Could not pack header, buf too small %d:%d\n", buf_len, res);
    return -1;
}
return res;
}

//-----

void flags_init (int argc, char *argv[]) {
    int i;
    for (i=1; i<argc; i++) {
        if (! strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (! strcmp(argv[i], "-?")) { usage(); /* does not return */ }
        if (! strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (! strcmp(argv[i], "-c") && (i+1 < argc)) {
            flag_i = 1; arg_inifile = argv[++i]; continue;
        }
        fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
        usage();
    }

    if (flag_i && arg_inifile) {
        if (strlen(arg_inifile) < sizeof(inifile)) {
            sprintf(inifile, "%s", arg_inifile);
        }
    }

    printf("Inifile is --> %s\n", inifile);
    printf("Debug is --> %d\n", flag_D);
    printf("Verbose is --> %d\n", flag_v);
    fflush(NULL);
}

```

```

//-----

void *handle_spectrometer(void *arg) {
//int unit_number = ((struct spec_object*)arg)->this_unit;
struct spec_object *myspec = (void *)arg;

int unit_number = myspec->this_unit; // quick reference

// Spectrometer
JSTRING spectrometerName;
JSTRING serialNumber;
JSTRING firmwareVersion;
double wavelengthIntercept;
double wavelengthCoefficient1;
double wavelengthCoefficient2;
double wavelengthCoefficient3;
int minimumAllowedIntegrationTime; // units: microseconds

int index;

int numberOfPixels;

DOUBLEARRAY_T spectrumArrayHandle;
double* spectrumValues;
DOUBLEARRAY_T wavelengthArrayHandle;

double* wavelengthValues;

// mosquito
struct mosquito *mosq;
int rc;

#define MQ_BUF_SIZ 256 * 1024
#define MQ_BUF 1024
#define SM_BUF 256

char mq_buf[MQ_BUF_SIZ]; // 4K pixel values with 5.2float + comma

char header_buf[SM_BUF];
char data_buf[MQ_BUF_SIZ];
int dl; // data length calculation
int ml; // message length


serialNumber = JString_Create();
spectrometerName = JString_Create();
firmwareVersion = JString_Create();

spectrumArrayHandle = DoubleArray_Create();
wavelengthArrayHandle = (DOUBLEARRAY_T)DoubleArray_Create();

printf("Handler %d Init\n",unit_number);

Wrapper_getSerialNumber(wrapperHandle,unit_number,serialNumber);
Wrapper_getName(wrapperHandle,unit_number,spectrometerName);
Wrapper_getFirmwareVersion(wrapperHandle,unit_number,firmwareVersion);

printf("Handler %d Spec type: %s  SN: %s  FW Vers: %s\n", unit_number,
JString_getASCII(spectrometerName),
JString_getASCII(serialNumber),
JString_getASCII(firmwareVersion));

// Demonstrate how to obtain and display wavelength coefficients
wavelengthIntercept = Wrapper_getWavelengthIntercept(wrapperHandle,unit_number);
wavelengthCoefficient1 = Wrapper_getWavelengthFirst(wrapperHandle,unit_number);
wavelengthCoefficient2 = Wrapper_getWavelengthSecond(wrapperHandle,unit_number);
wavelengthCoefficient3 = Wrapper_getWavelengthThird(wrapperHandle,unit_number);

printf("Wavelength intercept [%d] = %5.20f\n",unit_number,wavelengthIntercept);
printf("Coefficient 1st   [%d] = %.20f\n",unit_number,wavelengthCoefficient1);

```

```

printf("Coefficient 2nd    [%d] = %.20f\n",unit_number,wavelengthCoefficient2);
printf("Coefficient 3rd    [%d] = %.20f\n",unit_number,wavelengthCoefficient3);
printf("Integration time uS [%d] = %d\n",unit_number,myspec->integrationTime);
printf("Box Car Width      [%d] = %d\n",unit_number,myspec->boxCarWidth);
printf("Scan per avg       [%d] = %d\n",unit_number,myspec->scansAve);

// Set some default acquisition parameters
minimumAllowedIntegrationTime = Wrapper_getMinimumIntegrationTime(wrapperHandle,unit_number);
if (minimumAllowedIntegrationTime < myspec->integrationTime) {
    VPRINT(("Spectrometer[%d] Min Integration = %d\n",unit_number,minimumAllowedIntegrationTime));
    Wrapper_setIntegrationTime(wrapperHandle,unit_number,myspec->integrationTime);
}
else {
    fprintf(stderr,"Integration time too low Spec[%d],%d:%d\n",
        unit_number,
        minimumAllowedIntegrationTime,
        myspec->integrationTime);
}
Wrapper_setBoxcarWidth(wrapperHandle,unit_number,myspec->boxCarWidth);
Wrapper_setScansToAverage(wrapperHandle,unit_number,myspec->scansAve);
mosquitto_lib_init();
mosq = mosquitto_new(myspec->clientid, true, myspec->mosq_ptr);
if (mosq) {
    // mosquitto_connect_callback_set(mosq, connect_callback);
    // mosquitto_message_callback_set(mosq, message_callback);
    // start connection to mosquitto
    printf("Spec[%d].sbroker %s:%d, live %d\n",unit_number,
        myspec->sbroker, myspec->port, myspec->keepalive);

    rc = mosquitto_connect(mosq, myspec->sbroker, myspec->port, myspec->keepalive);
    if (rc) {
        printf("Spectrometer %d Link DOWN on %s [%d]\n",unit_number,myspec->queueName,rc);
        myspec->link_status = LINK_DOWN;
    }
    else {
        printf("Spectrometer %d Link UP on %s\n",unit_number,myspec->queueName);
        myspec->link_status = LINK_UP;
    }
}

// Start network worker bee
mosquitto_loop_start(mosq);

while (keepRunning) {

    usleep(MYSLEEP);

// set control

    if (myspec->parameterChange) {
        printf("Set New parameters to Spectrum Analyzer !\n");
        Wrapper_setIntegrationTime(wrapperHandle,unit_number,myspec->integrationTime);
        Wrapper_setBoxcarWidth(wrapperHandle,unit_number,myspec->boxCarWidth);
        Wrapper_setScansToAverage(wrapperHandle,unit_number,myspec->scansAve);
        Wrapper_setCorrectForElectricalDark(wrapperHandle,unit_number,myspec->electricalDark);
        myspec->parameterChange = 0; // done !
    }

// take a scan

    Wrapper_getSpectrum(wrapperHandle,unit_number,spectrumArrayHandle); // Acquire
    spectrumValues = DoubleArray_getDoubleValues(spectrumArrayHandle); // copy over array
    numberOfPixels = DoubleArray_getLength(spectrumArrayHandle); // find length of array

    if (unlikely((numberOfPixels > MAX_PIXELS)) {
        fprintf(stderr,"%s%d Too many Pixels %d\n",myspec->queueName,unit_number,numberOfPixels);
        numberOfPixels = MIN_PIXELS;
    }

    if(flag_D) {
        Wrapper_getWavelengths(wrapperHandle,unit_number,wavelengthArrayHandle);
        wavelengthValues = DoubleArray_getDoubleValues(wavelengthArrayHandle);

        // Loop to print the spectral data to the screen

```

```

for (index = 0; index<numberOfPixels; index++) {

    // pretty
    // printf("pixel %4d = %5.2f wavelength(nm): %1.2f\n",
    //     index,
    //     spectrumValues[index],
    //     wavelengthValues[index]);

    // comma delimited for spreadsheet import and display
    // printf("pixel,%4d,wavelength(nm),%1.2f,value,%5.2f\n",
    //     index,
    //     wavelengthValues[index],
    //     spectrumValues[index]);

    // really short ! .. no text
    printf("%4d,%1.2f,%5.2f\n",index,wavelengthValues[index],spectrumValues[index]);
}
}
if (0.0 != myspec->pixel_threshold) {
    // the threshold flag is set,
    // so compare pixel values to threshold to see
    // if we have some light
    // else, skip this sample and move on

    for (index = 0; index<numberOfPixels; index++) {
        if (spectrumValues[index] > myspec->pixel_threshold) {
            goto gt_threshold; // first past post
        }
    }
    goto done; // all failed to be over threshold
}

gt_threshold:

// buffer format is
// wavelength intercept, wavelength coeff 1,2,3, value[0..number of pixels -1]

dl = snprintf(data_buf, MQ_BUF, "%.20f,%.20f,%.20f,%.20f",
    wavelengthIntercept,
    wavelengthCoefficient1,
    wavelengthCoefficient2,
    wavelengthCoefficient3);

// sanity
if (unlikely(dl <= 0)) {
    fprintf(stderr, "Internal error data_buf 1 [%d] (%s)\n",dl,myspec->queuename);
    continue;
}

for (index = 0; index<numberOfPixels; index++) {
    dl += sprintf(&data_buf[dl], "%.2f", spectrumValues[index]);
    if (unlikely(dl >= (MQ_BUF_SIZ - 1024))) {
        fprintf(stderr, "Internal error data_buf 2 [%d] (%s)\n",dl,myspec->queuename);
        continue;
    }
}

// sprintf Will put one null as last char, so ok for printf!
if(flag_v)
    printf("mq->%s\n",data_buf);

build_shaper_mq_header(header_buf,
    SM_BUF,
    myspec->sensormname,
    "CSV",
    "float",
    numberOfPixels,
    "\\",
    dl);

ml = snprintf(mq_buf, MQ_BUF_SIZ, "%s:%s:%03d:", header_buf,data_buf,0xff);

```



```

    calc_tail(mq_buf, ml);

switch (myspec->link_status) {
case LINK_UP:
    rc = mosquitto_publish(mosq, NULL, myspec->queueName, strlen(mq_buf), mq_buf, 0, false);
    if (rc) { // not good !
        myspec->link_status = LINK_DOWN;
        printf("Spectrometer %d Link DOWN on %s [%d]\n", unit_number, myspec->queueName, rc);
    }
    break;
case LINK_DOWN:
    rc = mosquitto_reconnect(mosq); // attempt reconnect if broker went down
    if (!rc) {
        myspec->link_status = LINK_UP;
        printf("Spectrometer %d Link UP on %s\n", unit_number, myspec->queueName);
    }
    break;
default:
    printf("Internal error on link status on Spec[%d] %s\n", unit_number, myspec->queueName);
    break;
}

done:

;

} // while (keep running)

    JString_Destroy(firmwareVersion);
    JString_Destroy(serialNumber);
    JString_Destroy(spectrometerName);

    DoubleArray_Destroy(spectrumArrayHandle);
    DoubleArray_Destroy(wavelengthArrayHandle);

    mosquitto_disconnect(mosq);
    mosquitto_destroy(mosq);
    printf("Disconnected from broker\n");
    fflush(NULL);

    return NULL;
}

//-----

int main(int argc, char *argv[]) {

pthread_t thread[MAX_SPEC];
int iret[MAX_SPEC];

struct spec_object spectrometer[MAX_SPEC]; // array of spectrometers

// core
JSTRING apiVersion;
int buildNumber;
int numberOfSpectrometersAttached; // actually attached and talking to us
int spectrometerIndex;

    flags_init(argc, argv); // what did the user want
    load_ini_parms(inifile, spectrometer, MAX_SPEC); // read some ini parms

// Create the objects
    wrapperHandle = Wrapper_Create();
    apiVersion = JString_Create();

    buildNumber = Wrapper_getBuildNumber(wrapperHandle);

```

```

    Wrapper_getApiVersion(wrapperHandle,apiVersion);

    printf("Build number --> %d\n",buildNumber);
    printf("API version --> %s\n",JString_getASCII(apiVersion));

    // The following call will populate an internal array of spectrometer objects
    numberOfSpectrometersAttached = Wrapper_openAllSpectrometers(wrapperHandle);
    printf("# Spec found --> %d\n",numberOfSpectrometersAttached);
    if (numberOfSpectrometersAttached == 0)
    return EXIT_SUCCESS;

if (numberOfSpectrometersAttached > MAX_SPEC ) // limit the number !
    numberOfSpectrometersAttached = 4;

for (spectrometerIndex = 0; spectrometerIndex<numberOfSpectrometersAttached; spectrometerIndex++) {
    printf("Attempt attach Spectrometer %d to handler\n",spectrometerIndex);

    spectrometer[spectrometerIndex].this_unit = spectrometerIndex;
    spectrometer[spectrometerIndex].parameterChange = 1;
    spectrometer[spectrometerIndex].integrationTime = 85000;
    spectrometer[spectrometerIndex].boxCarWidth = 0;
    spectrometer[spectrometerIndex].scansAve = 5;
    spectrometer[spectrometerIndex].electricalDark = 1;
    spectrometer[spectrometerIndex].pixel_threshold = 0.0; // no thresholding

    // set any more objects here
    iret[spectrometerIndex] = pthread_create (&thread[spectrometerIndex],
        NULL,
        handle_spectrometer,
        (void *)&spectrometer[spectrometerIndex]);

    if (iret[spectrometerIndex]) {
        fprintf(stderr,"Error could not create worker thread %d\n",spectrometerIndex);
        exit (EXIT_FAILURE);
    }
    printf("Attach Spectrometer %d to handler %ld\n",
        spectrometerIndex,
        thread[spectrometerIndex]);
    sleep(1); // make sense of the output !
}

printf("Now Parent goes to sleep, while workers bee\n");

if (signal(SIGINT, intHandler) == SIG_ERR) {
    fprintf(stderr,"\n Cannot catch 'SIGINT'\n");
}
if (signal(SIGTERM, intHandler) == SIG_ERR) {
    fprintf(stderr,"\n Cannot catch 'SIGTERM'\n");
}

while (keepRunning) sleep(1); // attach Parser here .. TODO

printf("Sleeping Parent awakened, looks like we are closing down !\n");

// wait for children
for (spectrometerIndex = 0; spectrometerIndex<numberOfSpectrometersAttached; spectrometerIndex++) {
    printf("Attempt Detach Spectrometer %d from handler and await Joins\n",spectrometerIndex);
    pthread_join(thread[spectrometerIndex], NULL);
}

printf("Join threads\n");
fflush(NULL);

// Clean up
JString_Destroy(apiVersion);
Wrapper_closeAllSpectrometers(wrapperHandle);
Wrapper_Destroy(wrapperHandle);

printf("All Done\n");
fflush(NULL);

return EXIT_SUCCESS;
}

```


Appendix M Software for Monitoring the Magnetron Radio Emission Spectrum of Plasma Systems under Linux®

This code drives the SignalHound SA44B for monitoring the magnetron generator for the Hitachi 712 ECR plasma etcher system.

```
// sample compile is
// g++ -Wall -O2 rf_sa.cpp libSHAPPLIB.a libftd2xx.a -lpthread -ldl -lrt
// sudo ./a.out

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <pthread.h>
#include <signal.h>
#include <sys/time.h>
#include <errno.h>
#include <netdb.h>
#include "MySignalHound.h"
#include <mosquitto.h>

// # defines etc
#define VERSION "Version 0.1 "
#define MQTT_PORT 1883
#define LOCALHOST 0x7f000001
#define LINK_DOWN 0
#define LINK_UP 1
#define SCHED_PRIO 90
#define DEFAULT_CORE 1 // 0,1,2,3,4 ... (how many cores on machine ?) (i5 here)
// later .. will calculate intersect on machine

#define MYSLEEP 75000 // number of microseconds to sleep between 'sweeps'

char myhost[256] = "localhost";
char myport[16] = "1883";
char mymq_queue[64] = "RF";
char dBm_txt_buffer[128*1024]; // holds txt to send over MQ
char *arg_host = myhost;
char *arg_port = myport;
char *arg_ratio = NULL;
char *user_ptr = NULL;
char *arg_queue = mymq_queue;
int host = LOCALHOST;
int port = MQTT_PORT;
int link_status = LINK_DOWN;

volatile int keepRunning = 1;

// usage flags
int flag_h = 0;          /* host */
int flag_p = 0;          /* port */
int flag_v = 0;          /* verbose */
int flag_s = 0;          /* short format on MQ ! */
int flag_n = 0;          /* my queue name */
int flag_D = 0;          /* debug */

/*-----*/
long long current_timestamp() {
    struct timeval te;
    gettimeofday(&te, NULL); // get current time (deprecated..)
    long long milliseconds = te.tv_sec*1000LL + te.tv_usec/1000; // calculate milliseconds
    // printf("%s milliseconds: %lld\n", mymq_queue, milliseconds);
    return milliseconds;
}
/*-----*/
```

```

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d\n", (void*) &mosq, obj, result);
}
/*-----*/
void message_callback(struct mosquitto *mosq, void *obj, const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n", (void *) &mosq, (void *) &obj, (void *) &message);
}
/*-----*/
void intHandler(int dummy) {
    printf("\n%s sigint produced -> %d\n", mymq_queue, dummy);
    printf("%s Marking System for close, please be patient \n", mymq_queue);
    keepRunning = 0;
}
/*-----*/
void usage()
{
    printf("Berkeley Intel Spectrum Analyzer_mq \n");
    printf("rfsa_mq [-h host] [-D] [-v] [-n queue name] \n\n");
    exit(0);
}
/*-----*/
int getNumberOfCpus(void) {
    long nprocs = -1;
    long nprocs_max = -1;
    #ifdef _SC_NPROCESSORS_ONLN
    nprocs = sysconf(_SC_NPROCESSORS_ONLN);
    if ( nprocs < 1 ) {
        fprintf(stderr, " Could not determine number of CPUs online %s\n", strerror(errno));
        exit(0);
    }
    nprocs_max = sysconf(_SC_NPROCESSORS_CONF);
    if ( nprocs_max < 1 ) {
        fprintf(stderr, " Could not determine number of CPUs in host %s\n", strerror(errno));
        exit(0);
    }
    printf("Number of CPUs is --> %ld\n", nprocs_max);
    return nprocs;
#else
    fprintf(stderr, "Could not determine number of CPUs\n");
    return 0;
#endif
}
/*-----*/
void flags_init (int argc, char *argv[]) {
    int i;
    for (i=1; i<argc; i++) {
        if (! strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (! strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (! strcmp(argv[i], "-s")) { flag_s = 1; continue; }
        if (! strcmp(argv[i], "-h") && (i+1 < argc)) {
            flag_h = 1; arg_host = argv[i+1]; continue;
        }
        if (! strcmp(argv[i], "-p") && (i+1 < argc)) {
            flag_p = 1; arg_port = argv[i+1]; continue;
        }
        if (! strcmp(argv[i], "-n") && (i+1 < argc)) {
            flag_n = 1; arg_queue = argv[i+1]; continue;
        }
        fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
        usage();
    }
    if (flag_n && arg_queue) {
        sprintf(mymq_queue, "%s", arg_queue);
    }
    if (flag_p && arg_port) {
        port = atoi(arg_port);
        printf("port: = %d\n", port);
    }
    if (flag_h && arg_host) {
        struct hostent *hp;
        hp = gethostbyname(arg_host);
        if (! hp) {
            fprintf(stderr, "%s unknown host \"%s\"\n", mymq_queue, arg_host);
            exit(1);
        }
    }
}

```

```

    //printf("host-a: %08x\n", host);
    host = 0;
    for (i=0; i<4; i++) {
        int octet = hp->h_addr[i] & 0xff;
        //printf("%d\n", octet);
        host = (host << 8) | octet;
    }
    //printf("host-b: %08x\n", host);
}
printf("Broker is      --> %d.%d.%d.%d : %d\n\r",
      (host >> 24) & 0xff, (host >> 16) & 0xff,
      (host >> 8) & 0xff, (host >> 0) & 0xff, port);
printf("Queue is      --> %s\n", mymq_queue);
printf("Debug is      --> %d\n", flag_D);
printf("Verbose is    --> %d\n", flag_v);
printf("Short MQ is   --> %d\n", flag_s);

fflush(NULL);
}
/*-----*/

// using namespace std;
int main(int argc, char *argv[]) {

    int errorcode;
    int i,sweep_count;
    CMySignalHound myHound;
    double temperature_now;
    double centrefreq = 2450.0e6;    // centre frequency
    // double centrefreq = 915.0e6;    // centre frequency
    double span = 20.0e6;            // span to look at
    double ref_level = 0.0;          // reference level
    double RBW = 25.0e3;              // Resolution Bandwidth
    double HzPerPt = 0;              // The number of Hertz between samples
    double peak_freq = 0.0;
    double peak_ampl = 0.0;
    char debugstr[128];
    int peak_idx=0;
    int mos_res;
    int res;
    char client_id[24];
    char mq_buf[256 + (128*1024)];
    struct mosquitto *mosq;
    cpu_set_t *cpusetp, *maskp;
    size_t size;
    int num_cpus, cpu;
    sched_param schedparam;
    int ptr = 0;

    // begin .. say hello

    printf("\nBerkeley BioScientific : Plasma RF Spectrum Analyzer\n");
    printf("%s %s : %s\n", VERSION, __DATE__, __TIME__);
    flags_init(argc, argv);

    // Set the scheduling mode and priority
    schedparam.sched_priority = SCHED_PRIO;
    if (sched_setscheduler(0, SCHED_RR, &schedparam) != 0) {
        perror("Failed to set SCHED_RR and priority");
        exit(EXIT_FAILURE);
    }
    printf("Sched Prio is      --> %d\n", schedparam.sched_priority);

    // and set affinity to one of the available processors
    num_cpus = getNumberOfCpus();
    cpusetp = CPU_ALLOC(num_cpus); // will want the intersection of these two
    maskp = CPU_ALLOC(num_cpus); // but for moment will set to CPU 0

    if ((cpusetp == NULL) || (maskp == NULL)) {
        perror("CPU_ALLOC");
        exit(EXIT_FAILURE);
    }
    size = CPU_ALLOC_SIZE(num_cpus);

```

```

CPU_ZERO_S(size, cpusetp);

for (cpu = 0; cpu < num_cpus; cpu += 2)
    CPU_SET_S(cpu, size, cpusetp);
printf("CPU_COUNT() of set is --> %d\n", CPU_COUNT_S(size, cpusetp));

CPU_ZERO_S(size, maskp); // first clear the mask
CPU_SET_S(DEFAULT_CORE, size, maskp); // lets start with first cpu for moment

errorcode = sched_setaffinity(getpid(), size, maskp);
if (errorcode) {
    perror("Failed to set Processor Affinity");
    exit(EXIT_FAILURE);
}

printf("Affinity          --> %d : %d\n", errorcode, DEFAULT_CORE);

// Allow for clean exit
signal(SIGINT, intHandler); // assign signal handler
signal(SIGTERM, intHandler);

// Initialize the MQ publishing
mosquitto_lib_init();
memset(client_id, 0, 24);
snprintf(client_id, 23, "rf_spect_%d", getpid());
mosq = mosquitto_new(client_id, true, user_ptr);
if (mosq) {
    mosquitto_connect_callback_set(mosq, connect_callback);
    mosquitto_message_callback_set(mosq, message_callback);
    mos_res = mosquitto_connect(mosq, arg_host, port, 60);
    if (mos_res) {
        printf("%s Link Down %d\n", mymq_queue, mos_res);
        link_status = LINK_DOWN;
    }
    else {
        printf("%s Link UP\n", mymq_queue);
        link_status = LINK_UP;
    }
}
}
// Start network worker bee
mosquitto_loop_start(mosq);

// Initialize the Radio
if (flag_v)
    printf(">>> Attempt Radio Init\n");
errorcode = myHound.Initialize();
switch(errorcode) {
    case 0:
        sprintf(debugstr, ">>> Init Success\n");
        break;
    case 1:
        sprintf(debugstr, ">>> No Radio found .. Connect and retry %d\n",
            errorcode);
        break;
    case 2:
        sprintf(debugstr,
            ">>> Port did not open, FTDI found radio, but unable to open 'ports' %d\n",
            errorcode);
        break;
    case 98:
        sprintf(debugstr, ">>> Missing Temperature correction file 'D<sn>.bin' %d\n",
            errorcode);
        break;
    case 99:
        sprintf(debugstr, ">>> Missing calibration constants file '<sn>.tep' %d\n",
            errorcode);
        break;
    default:
        sprintf(debugstr, ">>> Unknown Error code returned for initialization %d\n",
            errorcode);
        break;
}
if (errorcode || flag_v) {
    printf("%s", debugstr);
}
}

```

```

if (errorcode) {
    goto tidy_up;
}
temperature_now = myHound.ReadTemperature();
errorcode = myHound.SetCenterAndSpan(centrefreq, span);
if (flag_v) {
    printf("Set Centre / Span returned %d\n",errorcode);
}
myHound.m_settings.m_RBWIsAuto = true;
myHound.m_settings.m_VBWIsAuto = true;
ref_level = 0.0; // for antennae .. later get from argv[]
myHound.m_settings.m_refLevel = ref_level;
errorcode = myHound.PreamplifierOnOff(true);
if (flag_v) {
    printf("Preamp on returned %d\n",errorcode);
}
errorcode = myHound.SetupForSweep();
if (flag_v) {
    printf("Setup for sweep returned %d\n",errorcode);
}
RBW = GetRBWFromIndex(myHound.m_settings.m_RBWSetspoint);
sprintf(debugstr, "RBW %.1f KHz", RBW * 1.0e-3);
if (RBW<1.0e3)
    sprintf(debugstr, "RBW %.1f Hz", RBW);
errorcode = myHound.DoSweep(); // perform a sweep to get RBW & tracesize
// wait for data on USB
usleep(MYSLEEP * 2);
if (flag_v || errorcode) { // the sweep can sometimes return error,
    // so override here to spit out
    printf("First Sweep returned %d\n",errorcode);
}
HzPerPt = myHound.m_HzPerPt; // just copy for moment
printf("Ref Level is      --> %f\n", ref_level);
printf("TraceSize is      --> %d\n", myHound.m_traceSize);
printf("Resolution Bandwidth --> %s\n", debugstr);
printf("Temperature is    --> %f\n", temperature_now);
printf("Hertz Per Point is --> %f\n", HzPerPt);
// ok .. main loop

sweep_count = 0;
dBm_txt_buffer[0]=0;
while (keepRunning) {
    ++sweep_count &= 0xff; // lets keep it sane
    if ((sweep_count / 10) * 10 == sweep_count) { // every 10 or so as reqd
        temperature_now = myHound.ReadTemperature(); // to update temperature calib
        if (flag_v)
            printf("Temperature is      --> %f\n", temperature_now);
    }
}
// Note !!!! NOT thread safe
errorcode = myHound.DoSweep(); //Do the sweep .. this shud be a blocking function
// but unfortunately returns before data ready
usleep(MYSLEEP); // on i5 .. anything < 60mS and next sweep returns error
if (errorcode || flag_v) {
    printf("Sweep %d returned %d\n",sweep_count,errorcode);
}
if (errorcode) {
    usleep(MYSLEEP);
    continue; // no point .. data not valid / incomplete
}
// search for peak
peak_idx = 0;
ptr = 0; // point at the start .. we will bump as print in each value
for (i = 1; i < myHound.m_traceSize; i++) {
    if (myHound.pDataMax[i] > myHound.pDataMax[peak_idx]) {
        peak_idx = i; // if current GT previous greatest .. store !
    }
}
// at the same time convert .. Only walk array minimum # times !!
if (!flag_s) {
    res = sprintf(&dBm_txt_buffer[ptr], "%.1f", mW2dBm(myHound.pDataMax[i]));
    if ((res > 0) && (ptr < (128 * 1024) - 2)) {
        ptr += res;
    }
} else {
    fprintf(stderr, "Internal allocation error on sprintf on convert %d:%d\n",res,ptr);
}

```



```

    }
}
peak_freq = myHound.GetFrequencyFromIdx(peak_idx); // get peak frequency
peak_ampl = mW2dBm(myHound.pDataMax[peak_idx]); // and peak amplitude

if (flag_v) {
    printf(">> sweep %d:%d got %d samples pk @ %d, %.1f dBm, @ %f Hz\n",
        sweep_count, errorcode, myHound.m_traceSize, peak_idx, peak_ampl, peak_freq);
}

#ifdef 0
// dump ( some of / ) the array for debug
// for (i = 0; i < myHound.m_traceSize; i++) {
//     for (i = 0; i < 10; i++) {
//         printf(">>> i -> %d : %f (%f dBm) : %f (%f dBm) \n", i,
//             myHound.pDataMax[i], mW2dBm(myHound.pDataMax[i]),
//             myHound.pDataMin[i], mW2dBm(myHound.pDataMin[i]));
//     }
// }
#endif

#ifdef 0
// some stats for debug
printf("dBm_txt_buffer -> %p\n", dBm_txt_buffer);
printf("ptr -> %p\n", ptr);
for (i = 0; i < 40; i++) {
    printf("%c", dBm_txt_buffer[i]);
}
printf("\n^txt buffer !\n");
#endif

// lets post on to the broker

if (flag_s) {
    sprintf(mq_buf, "%lld,%x,%d,%d,%.1f,%.0f",
        current_timestamp(),
        sweep_count,
        myHound.m_traceSize,
        peak_idx, peak_ampl, peak_freq);
}
else {
    sprintf(mq_buf, "%lld,%x,%d,%d,%.1f,%.0f%s",
        current_timestamp(),
        sweep_count,
        myHound.m_traceSize,
        peak_idx, peak_ampl, peak_freq,
        dBm_txt_buffer);
}

switch (link_status) {
case LINK_UP:
    mos_res = mosquitto_publish(mosq, NULL, mymq_queue, strlen(mq_buf), mq_buf, 0, false);
    if (mos_res) { // not good !
        link_status = LINK_DOWN;
        printf ("%s -> link down \n", mymq_queue);
    }
    break;
case LINK_DOWN:
    mos_res = mosquitto_reconnect(mosq); // attempt reconnect if broker went down
    if (!mos_res) {
        link_status = LINK_UP;
        printf ("%s -> link up \n", mymq_queue);
    }
    break;
default:
    printf ("%s Internal error on link status\n", mymq_queue);
    break;
}
}

tidy_up:
mos_res = mosquitto_disconnect(mosq);
printf(">> MQ disconnect -> %d\n", mos_res);
errorcode = ~myHound.Initialize();

```

```
// seems to return -1 ??  
if (errorcode == -1) {  
    printf(">> DelInit Success -> %d\n", errorcode);  
}  
else {  
    printf(">> DelInit Fail -> %d\n", errorcode);  
    // exit(1);  
}  
CPU_FREE(cpusetp);  
CPU_FREE(maskp);  
printf("Bye\n");  
return 0;  
}
```

Appendix N Software for Monitoring the Daihen Autotuner on the Hitachi 712 ECR Plasma System operating on Linux®

This code drives an isolated USB interface to acquire data using the proprietary protocol of the Daihen microwave autotuner.

```
// #includes here
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <signal.h>
#include <sys/time.h>
#include <errno.h>
#include <netdb.h>
#include <termios.h>
#include "daihen.h"
#include "ftd2xx.h"
#include "wr3_time.h"

#include <mosquitto.h>

#include "iniparser.h"

// # defines etc

#define MQ_BUF 1024
#define SM_BUF 256

#define MQTT_PORT 1883
#define DEFAULT_RATIO 4
#define DEFAULT_THRESHOLD 10
#define LOCALHOST 0x7f000001
#define LINK_DOWN 0
#define LINK_UP 1
#define NUM_DEV 4

#define POWER_THRESHOLD 5
#define KEEPALIVE 60

#if 1
#define VPRINT(A) do { if (flag_v) { printf A; } } while (0)
#define VDPRINT(A) do { if (flag_v && flag_D) { printf A; } } while (0)
#else
#define VPRINT(A)
#define VDPRINT(A)
#endif

// for the database format for Q output

#define MQ_STR "'%s', cast('%s' as datetime),%.2f,%.2f,%.2f,%d,%d,%d,%d,%d,%f,%d,%f,%d,%f,%d"

// New Shaper MQ message format (see Specification Document)
// Header = "mymq_queue;wr3_date();CSV;float,int;15,;len;"
// Data = "S0,
//      S1,
//      S2,
//      Foward_power, Fwrdr_flag,
//      Reflected_power, Reflected_flag,
//      Uin, Uin_flag,
//      Vin, Vin_flag,
```

```

//      Uload, Uload_flag,
//      Yload, Yload_flag;"

#define DATA_FORMAT "%0.2f,%0.2f,%0.2f,%d,%d,%d,%d,%f,%d,%f,%d,%f,%d,%f,%d"

// globals

// Inifile stuff
char inifile[SM_BUF]="at0.ini";
char *arg_inifile = inifile;

// FT driver stuff !
FT_STATUS isolator_status;
FT_HANDLE isolator_handle = NULL;
// Isolator Serial Numbers
// char serial_number[SM_BUF] = "A4018G30";
// char serial_number[SM_BUF] = "A4013JUZ";
// char serial_number[SM_BUF] = "A4018GJ3";
// char serial_number[SM_BUF] = "A4018FDD";
char devname[SM_BUF]; // no default !

// Mosquitto
char sbroker[SM_BUF] = "localhost";
unsigned int broker = 0x7f000001; // localhost IPV4 only
char clientid[SM_BUF];
int mqport = MQTT_PORT;
char topicname[SM_BUF] = "ICE121/AT0";
char sensorname[SM_BUF] = "AT0";
int persist = false;
int keepalive = KEEPALIVE;

// Serial Stuff
int baud_rate = B9600;
int stop_bits = CSTOPB; // set two bits if set else 1
int parity = PARODD; // Odd parity else even
int parity_en = PARENB; // parity is enabled
int data_bits = CS8; // CS7 or CS8 ~ 7 or 8 bits
char delimiter = CR; // LF or CR or SC (on machine) but manual says CR,LF or NULL

char input_buffer[MQ_BUF]; // used by emulator to parse messages (could combine .. but at 256 bytes ?)
char output_buffer[MQ_BUF]; // used by master to push out

// Command line paramaters (now moved to ini file)

int ratio = DEFAULT_RATIO;
int forward_power_threshold = POWER_THRESHOLD; // the threshold -> state 0 to state 1
int link_status = LINK_DOWN;

volatile int keepRunning = 1;

// usage flags
int flag_t = 0; // /* Test mode */
int flag_v = 0; /* verbose */
int flag_D = 0; /* debug */
int flag_A = 0; /* All records even if haven't exceed Pwr Threshold */
int flag_i = 0; /* Did we get an ini file ? */
int flag_H = 0; /* Reserved for futuer use ! */

// constants
#define NO_OF_MESSAGES 30
char msgs[NO_OF_MESSAGES][3] = {"MF","MR","S0","S1", /* 0, 1, 2, 3 */
                                "S2","W0","W1","W2", /* 4, 5, 6, 7 */
                                "WT","WM","P0","P1", /* 8, 9, 10, 11 */
                                "P2","PT","PM","M0", /* 12, 13, 14, 15 */
                                "M1","M2","MU","MV", /* 16, 17, 18, 19 */
                                "LU","LV","SU","SV", /* 20, 21, 22, 23 */
                                "C0","C1","SA","RL", /* 24, 25, 26, 27 */
                                "RS","RB"); // /* 28, 29, 30, 31 */

// note .. you do not seem to be able to ENQ the messages RL RS or RB

```

```

#define MSG_MF 0
#define MSG_MR 1
#define MSG_S0 2
#define MSG_S1 3
#define MSG_S2 4
#define MSG_W0 5
#define MSG_W1 6
#define MSG_W2 7
#define MSG_WT 8
#define MSG_WM 9
#define MSG_P0 10
#define MSG_P1 11
#define MSG_P2 12
#define MSG_PT 13
#define MSG_PM 14
#define MSG_M0 15
#define MSG_M1 16
#define MSG_M2 17
#define MSG_MU 18
#define MSG_MV 19
#define MSG_LU 20
#define MSG_LV 21
#define MSG_SU 22
#define MSG_SV 23
#define MSG_C0 24
#define MSG_C1 25
#define MSG_SA 26
#define MSG_RL 27
#define MSG_RS 28
#define MSG_RB 29

// some microwave data ! (arbitrary values!)

int forward_wave_power = 0;           // range is 0 to 2400
int reflected_wave_power = 0;         // range is 0 to 2400
float stub_manual_setting[3] = {0.0,0.0,0.0}; // Range 00.00 to 25.00 for 3 stubs
float poff_preset_stub_setting[3] = {0.0,0.0,0.0}; // Range 00.00 to 25.00 for 3 stubs
float poff_preset_timer_setting = 0.0; // range is 0 to 60.0
int poff_preset_setting = 1;          // flag
float pon_preset_stub_setting[3] = {0.0,0.0,0.0}; // Range 00.00 to 25.00 for 3 stubs
float pon_preset_timer_setting = {0.0}; // range is 0 to 60.0
int pon_preset_setting = 1;           // flag
float stub_preset_position_reading[3] = {0.0,0.0,0.0}; // range 00.00 to 25.00
float input_reflection_coefficient_uin = 0.0; // range -1.0 to 1.0
float input_reflection_coefficient_vin = 0.0; // range -1.0 to 1.0
float load_reflection_coefficient_oload = 0.0; // range -1.0 to 1.0
float load_reflection_coefficient_yload = 0.0; // range -1.0 to 1.0
float set_reflection_coefficient_u = 0.0; // range -0.5 to 0.5
float set_reflection_coefficient_v = 0.0; // range -0.5 to 0.5
float set_dead_zone = 0.0; // range 0 to 0.5
int uwave_power_on_threshold = 0; // range 0 to 500
int auto_manual_tuning = 0; // range 0 or 1
int select_memory_load = 0; // range is 0 to 7
int select_memory_saved = 0; // range is 0 to 7
int present_set_value_backed_up = 0; // range is 0 to 1000 (note 4 p18)

// some flags to indicate the 'freshness' of values as we print
// them to the queue ... we could just print fixed values, but lets
// calculate them. This allows for expansion if we decide to read
// further value from the Daihen (or even use A/D !)
// This is a bit field
int flag_fresh = 0; //Nothing is fresh to start

int test_count = 0; // simulate power from a Daihen for bench tests
// we will generate a 'ramp' power

pthread_mutex_t timesync; // use a mutex for the timer to let the main run

//-----
// Typedefs

```

```

long long current_timestamp();

//=====================================================

void sigalrm_handler(int signum) {
    pthread_mutex_unlock(&timesync);
    if (flag_v && flag_D) {
        long long timecnt = current_timestamp();
        printf("SIGALRM received [%d], count :%lld\n",signum,timecnt);
    }
}

//=====================================================

void intHandler(int dummy) {
    printf("\n%s sigint produced -> %d\n",sensorname,dummy);
    printf("%s Marking System for close, please be patient \n",sensorname);
    keepRunning = 0;
}

//=====================================================

//-----
#define LO 0
#define HI 1

char tohex(unsigned char chr, int hilo) {
    if (hilo==HI)
        chr = (chr>>4) & 0x0f;
    else
        chr = chr & 0x0f;
    switch(chr) {
        case 0x0: return '0'; case 0x01: return '1'; case 0x02 : return '2'; case 0x03: return '3';
        case 0x4: return '4'; case 0x05: return '5'; case 0x06 : return '6'; case 0x07: return '7';
        case 0x8: return '8'; case 0x09: return '9'; case 0x0a : return 'a'; case 0x0b: return 'b';
        case 0xc: return 'c'; case 0x0d: return 'd'; case 0x0e : return 'e'; case 0x0f: return 'f';
    }
    return 0; // dummy
}

//-----

void emulate_power() { // just arbitrary
    test_count++;
    if (test_count == 120) // simulate that we see power over threshold
        test_count = 0; // every 120 or so
        // we will attack,sustain,decay, off and repeat
    if (test_count < 30) // start & attack
        forward_wave_power = test_count;
    if ((test_count >= 30) && (test_count < 80)) // sustain (climbing slightly !)
        forward_wave_power = (test_count) * 5;
    if ((test_count >= 80) && (test_count < 120)) // decay
        forward_wave_power = (120 - test_count) * 7;

    reflected_wave_power = forward_wave_power / 5;

    input_reflection_coefficient_uin = forward_wave_power / 100.0;
    input_reflection_coefficient_vin = 1.0-(forward_wave_power / 99.0);
    load_reflection_coefficient_uload = reflected_wave_power / 100.0;
    load_reflection_coefficient_yload = 2.0 - (reflected_wave_power / 99.0);

    // flag_fresh |= (FL_MF | FL_MR | FL_MU | FL_MV | FL_LU | FL_LV);

}

//-----

long long current_timestamp() {
    int res;

```

```

struct timespec ts;
res = clock_gettime(CLOCK_REALTIME, &ts); {
if (res)
perror("could not read clock");
strerror(errno);
}
long long milliseconds = ts.tv_sec*1000LL + ts.tv_nsec/1000000; // calc ms
return milliseconds;
}

//-----

int tstring(char *buf, int len) {
int res;
struct timespec ts;
struct tm t;

res = clock_gettime(CLOCK_REALTIME, &ts);
if (res) {
perror("could not read clock");
strerror(errno);
}

tzset();
if (localtime_r(&(ts.tv_sec), &t) == NULL) {
return 1;
}
res = strftime(buf, len, "%F %T", &t);

if (0 == res) {
return 2;
}
len -= res -1;

// nS reporting
res = snprintf(&buf[strlen(buf)], len, "%.09ld", ts.tv_nsec);

// want mS reporting
res = snprintf(&buf[strlen(buf)], len, "%.03ld", (ts.tv_nsec/1000000));

if (res >= len) {
return 3;
}

return 0;
}

//-----

int wait_half_sec(long long then) {
long sleep_us;
long elapsed;

// We want 2Hz rate -> .5 of a second between acquisitions
// so calc how long betwenn 'now' and then
// make up the time to .5 sec or return

elapsed = current_timestamp() - then; // milleseconds
sleep_us = 500000 /* uS */ - (elapsed * 1000);
if (flag_v)
printf("Elapsed = %ld mS, sleep = %ld\n", elapsed, sleep_us);
if ((sleep_us < 500000) && (sleep_us > 0)) {
usleep(sleep_us);
}
return(sleep_us);
}

//-----

int check_devices(void) {
FT_STATUS ftStatus;
char *bufptr[NUM_DEV + 1];
char sn[NUM_DEV][64]; // space for serial number
DWORD numdevs = 0;

```

```

int i,res;
int ret = 0;

// set up an array of pointers to string bufs for call to ListDevices
// for API .. last one is NULL
for(i = 0; i < NUM_DEV; i++) {
    bufptr[i] = sn[i];
    sprintf(sn[i], "%s", devname);
}
    bufptr[NUM_DEV] = NULL;

ftStatus = FT_ListDevices(bufptr, &numdevs , FT_LIST_ALL | FT_OPEN_BY_SERIAL_NUMBER);

printf("/*-----*\n");
printf("/* Searching for a list of available devices first *\n");
printf("/*-----*\n");

    if (ftStatus == FT_OK) {
        if (!numdevs) {
            printf("/* No device\n");
            ret = -1;
        }
        else {
            for (i = 0; i<(int)numdevs; i++) {
                printf("/* Device -> %d = %s\n",i,bufptr[i]);
// try opening the particular one that we are interested in !!
// The user specified the particular number.. so we will index
// into it !
                res = strcmp(bufptr[i],devname,min(sizeof(bufptr[i]),sizeof(devname)));
                if (!res) {
                    printf("/* Found an Isolator Autotuner station ! %d,%s\n",i,bufptr[i]);
                }
            }
        }
    }
    else {
        fprintf(stderr, "Error getting list of devices\n");
    }
    printf("/*-----*\n");
    printf("/* End of Search *\n");
    printf("/*-----*\n");
    return ret;
}

//-----

int setup_device(void) {
int baud = 9600;
int dbits = FT_BITS_8;
int sbits = FT_STOP_BITS_2;
int par = FT_PARITY_ODD;
ULONG stat;

retry_setup:
    if (!keepRunning) { // just in case no device, and SIGINT called
        return -1;
    }
// lets just check first whether we re-entered on 'retry'
    if (isolator_handle) {
        FT_Close(isolator_handle);
        if (flag_v) {
            printf("%s Closed device %s\n",sensorname, devname);
        }
        isolator_handle = NULL; // mark that it is closed
    }
    if (flag_v && flag_D) {
        printf("Attempting to open '%s' FT_OPEN_BY_SERIAL_NUMBER\n",devname);
    }
    isolator_status = FT_OpenEx(devname,
        FT_OPEN_BY_SERIAL_NUMBER, &isolator_handle);

    if (isolator_status != FT_OK){
        /*
This can fail if the ftdi_sio driver is loaded

```


use lsmod to check this and rmmod ftdi_sio to remove
also rmmod usbserial

```
*/
if (flag_v) {
    printf("%s Failed to open [%d] .. retry\n",sensorname, (int)isolator_status);
}
sleep(5);
goto retry_setup;
}
switch(baud_rate) { // just in case FT does not use Termios standards
    case B300: baud = 300; break;
    case B600: baud = 600; break;
    case B1200: baud = 1200; break;
    case B2400: baud = 2400; break;
    case B4800: baud = 4800; break;
    default: baud = 9600; break;
}
if (parity_en) {
    if (parity==PARODD) {
        par=FT_PARITY_ODD;
    }
    else {
        par=FT_PARITY_EVEN;
    }
}
else {
    par = FT_PARITY_NONE;
}
switch(stop_bits) {
    case CSTOPB: sbits = FT_STOP_BITS_2; break;
    default: sbits = FT_STOP_BITS_1; break;
}
switch(data_bits) {
    // case CS5: dbits = FT_BITS_5; break; // not supported ?
    // case CS6: dbits = FT_BITS_6; break;
    case CS7: dbits = FT_BITS_7; break;
    default: dbits = FT_BITS_8; break;
}

isolator_status = FT_SetBaudRate(isolator_handle, baud);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set baudrate [%d]\n",sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}

//isolator_status = FT_SetDataCharacteristics(isolator_handle, FT_BITS_8, FT_STOP_BITS_2,
FT_PARITY_ODD);
isolator_status = FT_SetDataCharacteristics(isolator_handle, dbits, sbits, par);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set line parameters [%d]\n",sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}

isolator_status = FT_SetTimeouts(isolator_handle,5000,0);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set timeouts [%d]\n",sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}

isolator_status = FT_SetFlowControl(isolator_handle,FT_FLOW_NONE, 0x11, 0x13);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set Flow Control [%d]\n",sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}
```

```

}

isolator_status = FT_SetDtr(isolator_handle);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set DTR [%d]\n", sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}

isolator_status = FT_SetRts(isolator_handle);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to set RTS [%d]\n", sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}

isolator_status = FT_GetModemStatus(isolator_handle, &stat);
if ((isolator_status) != FT_OK) {
    if (flag_v) {
        printf("%s Failed to get Modem Status [%d]\n", sensorname, (int)isolator_status);
    }
    sleep(1);
    goto retry_setup;
}
else {
    if (flag_v || flag_D) {
        printf("CTS -> %d\n", stat&0x0010?1:0);
        printf("DSR -> %d\n", stat&0x0020?1:0);
        printf("RI -> %d\n", stat&0x0040?1:0);
        printf("DCD -> %d\n", stat&0x0080?1:0);
        printf("OV -> %d\n", stat&0x0200?1:0);
        printf("PE -> %d\n", stat&0x0400?1:0);
        printf("FE -> %d\n", stat&0x0800?1:0);
        printf("BI -> %d\n", stat&0x1000?1:0);
    }
}

printf("%s Device UP\n", sensorname);
return(0);

}

//-----

int chk_range_f (char *buf, float *curr, float new, float min, float max, int fresh_bit) {
    if ((new < min) || (new > max)) {
        fprintf(stderr, "%s [%c%c] value out of range [%f - %f] %f\n", sensorname, buf[1], buf[2], min, max, new);
        return -1;
    }
    else {
        *curr = new; // update
        flag_fresh |= fresh_bit; // mark that we have updated this field just now
    }
    return 0;
}

//-----

int chk_range_i (char *buf, int *curr, int new, int min, int max, int fresh_bit) {
    if ((new < min) || (new > max)) {
        fprintf(stderr, "%s [%c%c] value out of range [%d - %d] %d\n", sensorname, buf[1], buf[2], min, max, new);
        return -1;
    }
    else {
        *curr = new;
        flag_fresh |= fresh_bit;
    }
    return 0;
}

```

```
//-----

int get_value(char *buf, int len) {
// suppose it would be nice to make it table driven !!
// 3 char header, data, etx, chksum, delim
#define MAX_VAL_SZ 8// just enough bytes to store
#define HDR_SZ 3 // STX + COMMAND0 + COMMAND1
#define TAIL_SZ 4 // ETX + CHK_HI + CHK_LO + DELIM
char tmp[MAX_VAL_SZ]; // don't see values much bigger than this
int i;
int res;
float tmp_f;
int tmp_i;
memset(tmp,0,sizeof(tmp));
if ((len-(HDR_SZ + TAIL_SZ) > MAX_VAL_SZ)) return -1; // too long .. something wrong
for (i=HDR_SZ; i<=len-TAIL_SZ; i++) {
    tmp[i-HDR_SZ] = buf[i];
}
switch(buf[1]) {
case 'M':
    if ((buf[2] == 'F') || (buf[2] == 'R')) {
        res=sscanf(tmp,"%d", &tmp_i);
        if (buf[2] == 'F')
            chk_range_i(buf,&forward_wave_power, tmp_i, POWER_MIN_I, POWER_MAX_I, FL_MF);
        if (buf[2] == 'R')
            chk_range_i(buf,&reflected_wave_power, tmp_i, POWER_MIN_I, POWER_MAX_I, FL_MR);
    }
    else {
        res=sscanf(tmp,"%f", &tmp_f);

        if (buf[2] == '0')
            chk_range_f(buf,&stub_preset_position_reading[0], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
            FL_M0);
        if (buf[2] == '1')
            chk_range_f(buf,&stub_preset_position_reading[1], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
            FL_M1);
        if (buf[2] == '2')
            chk_range_f(buf,&stub_preset_position_reading[2], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
            FL_M2);

        if (buf[2] == 'U')
            chk_range_f(buf,&input_reflection_coefficient_uin, tmp_f, BIG_COEFF_MIN_F, BIG_COEFF_MAX_F,
            FL_MU);
        if (buf[2] == 'V')
            chk_range_f(buf,&input_reflection_coefficient_vin, tmp_f, BIG_COEFF_MIN_F, BIG_COEFF_MAX_F,
            FL_MV);
    }
    break;
case 'S':
    if (buf[2] == 'A') {
        res=sscanf(tmp,"%d", &tmp_i);
        chk_range_i(buf,&auto_manual_tuning, tmp_i, SETTING_MIN_I, SETTING_MAX_I, FL_SA);
    }
    else {
        res=sscanf(tmp,"%f", &tmp_f);
        if (buf[2] == '0')
            chk_range_f(buf,&stub_manual_setting[0], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_S0);
        if (buf[2] == '1')
            chk_range_f(buf,&stub_manual_setting[1], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_S1);
        if (buf[2] == '2')
            chk_range_f(buf,&stub_manual_setting[2], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_S2);

        if (buf[2] == 'U')
            chk_range_f(buf,&set_reflection_coefficient_u, tmp_f, SM_COEFF_MIN_F, SM_COEFF_MAX_F, FL_SU);
        if (buf[2] == 'V')
            chk_range_f(buf,&set_reflection_coefficient_v, tmp_f, SM_COEFF_MIN_F, SM_COEFF_MAX_F, FL_SV);
    }
    break;
case 'W':
    if (buf[2] == 'M') {
        res=sscanf(tmp,"%d", &tmp_i);
        chk_range_i(buf,&poff_preset_setting, tmp_i, SETTING_MIN_I, SETTING_MAX_I, FL_WM);
    }
    else {
        res=sscanf(tmp,"%f", &tmp_f);

```

```

        if (buf[2] == '0')
            chk_range_f(buf,&poff_preset_stub_setting[0], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
FL_W0);
        if (buf[2] == '1')
            chk_range_f(buf,&poff_preset_stub_setting[1], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
FL_W1);
        if (buf[2] == '2')
            chk_range_f(buf,&poff_preset_stub_setting[2], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F,
FL_W2);
        if (buf[2] == 'T')
            chk_range_f(buf,&poff_preset_timer_setting, tmp_f, TIMER_MIN_F, TIMER_MAX_F, FL_WT);
    }
    break;
    case 'p':
        if (buf[2] == 'M') {
            res=sscanf(tmp,"%d", &tmp_i);
            chk_range_i(buf,&pon_preset_setting, tmp_i, SETTING_MIN_I, SETTING_MAX_I, FL_PM);
        }
        else {
            res=sscanf(tmp,"%f", &tmp_f);
            if (buf[2] == '0')
                chk_range_f(buf,&pon_preset_stub_setting[0], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_P0);
            if (buf[2] == '1')
                chk_range_f(buf,&pon_preset_stub_setting[1], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_P1);
            if (buf[2] == '2')
                chk_range_f(buf,&pon_preset_stub_setting[2], tmp_f, STUB_VALUE_MIN_F, STUB_VALUE_MAX_F, FL_P2);
            if (buf[2] == 'T')
                chk_range_f(buf,&pon_preset_timer_setting, tmp_f,TIMER_MIN_F, TIMER_MAX_F, FL_PT);
        }
    }
    break;
    case 'L':
        res=sscanf(tmp,"%f", &tmp_f);
        if (buf[2] == 'U')
            chk_range_f(buf,&load_reflection_coefficient_oload, tmp_f, BIG_COEFF_MIN_F, BIG_COEFF_MAX_F,
FL_LU);
        if (buf[2] == 'V')
            chk_range_f(buf,&load_reflection_coefficient_yload, tmp_f, BIG_COEFF_MIN_F, BIG_COEFF_MAX_F,
FL_LV);
    }
    break;
    case 'C':
        if (buf[2] == '0') {
            res=sscanf(tmp,"%f", &tmp_f);
            chk_range_f(buf,&set_dead_zone, tmp_f, DZ_MIN_F, DZ_MAX_F, FL_C0);
        }
        else {
            if (buf[2] == '1')
                res=sscanf(tmp,"%d", &tmp_i);
            chk_range_i(buf,&uwave_power_on_threshold, tmp_i, THRESHOLD_MIN_I, THRESHOLD_MAX_I, FL_C1);
        }
    }
    break;
    default:
        res = -1;
    break;
}
if (res < 1) {
    fprintf(stderr, "%s Error decoding value for %c%c\n",sensorname,buf[1],buf[2]);
    return -2;
}
return 0;
}

//-----

int calc_checksum(int etx_locn, char *buf, char *hi, char *lo) {
    unsigned char checksum = 0;
    int i;
    if (!(buf[etx_locn] == ETX) && ((buf[0] == STX) || (buf[0] == ENQ))) return -1;
    for (i=1; i<etx_locn; i++) {
        checksum += buf[i];
    }
    *hi=0x30 | ((checksum & 0xf0) >> 4);
    *lo=0x30 | (checksum & 0x0f);
    return 1;
}

```

```

//-----

int test_msg_format (char *buf, int len) {
char hi, lo;
if ((buf[0] != STX) || (buf[len-3] != ETX)) return -1;
calc_checksum(len-3,buf, &hi, &lo);
if ((hi != buf[len-2]) || (lo != buf[len-1])) return -2;
return 0;
}

//-----

void enq_msg(char *type, char *buf, int delim) {
// fixed length, insert message, checksum and delimiter
//      0123 4567
printf(buf,"%05%s\003 ",type);
#define ENQ_ETX_LOCN 3
calc_checksum(ENQ_ETX_LOCN,&buf[0],&buf[4],&buf[5]);
buf[6] = (char) delim;
}

//-----

int process_msg(int msg_index) {
// We stay here and only proceed if the device is online
// and we can read/assemble a message
// In all cases of error, we just wait for the
// device to come online again as there is not much we can do without
int res;
int ptr;
int i;
int timeout;
DWORD dwBytesRead;
DWORD dwBytesWritten;

retry_process:

if (!isolator_handle) { // The device is up and running ?
res = setup_device();
if (res)
return -1;
}
if (flag_v) {
printf("\n%s Send ENQ\n",sensorname);
}

enq_msg(msgs[msg_index], output_buffer, delimiter);
if (flag_v && flag_D) {
printf("ENQ is -> ");
for (i=0; i< ENQ_LENGTH; i++)
printf("[%0.2x] ",output_buffer[i]);
printf("\n");
}

isolator_status = FT_Write(isolator_handle, output_buffer, ENQ_LENGTH, &dwBytesWritten);
if ((isolator_status != FT_OK) || (dwBytesWritten != ENQ_LENGTH)) {
if (flag_v) {
printf("\n%s Could not send ENQ\n",sensorname);
}
sleep(1);
FT_Close(isolator_handle);
isolator_handle = NULL;
printf("%s Device DOWN\n",sensorname);
goto retry_process;
}

if (flag_v)
printf("\n%s Read isolator\n",sensorname);
ptr = 0; timeout = 0;
memset(input_buffer, 0, sizeof(input_buffer));
while ((timeout < 2) && keepRunning) { // 2 times 5000 mS --> ~10 seconds
isolator_status = FT_Read(isolator_handle, &input_buffer[ptr], 1, &dwBytesRead);
if ((isolator_status == FT_OK) && (dwBytesRead == 1)) {
if (flag_v)
printf("%s Got char %d:[%c%c]\n",sensorname,

```

```

        ptr,tohex(input_buffer[ptr],HI),tohex(input_buffer[ptr],LO));
    if (input_buffer[ptr] == (char) delimiter) {
        input_buffer[ptr+1] = 0;
        goto input_end;
    }
    ptr++;
}
else {
    // printf(".");
    fflush(NULL);
    timeout++;
}
}
if (timeout == 2) { // have we run out of timeouts ??
    if (flag_v) {
        printf("\n%s Oops .. timeout on input\n",sensorname);
    }
    sleep(1);
    FT_Close(isolator_handle);
    isolator_handle = NULL;
    printf("%s Device DOWN\n",sensorname);
    goto retry_process;
}
input_end:
// ptr is length -> points to delimiter
if ((res=test_msg_format(input_buffer, ptr)) == 0) {
    if (flag_v)
        printf("%s Chk_msg was:%d\n",sensorname,res);
    if (get_value(input_buffer,ptr))
        printf("%s Did not get value\n",sensorname);
}
else
    return -1;
return 0;
}

//-----

void connect_callback(struct mosquitto *mosq, void *obj, int result) {
    printf("-> Connect callback %p,%p,%d!\n",(void*) &mosq,obj,result);
}

//-----

void message_callback(struct mosquitto *mosq, void *obj, const struct mosquitto_message *message) {
    printf("-> message callback %p,%p,%p!\n",(void *) &mosq, (void *) &obj, (void *) &message);
}

//-----

void usage() {
    printf("Berkeley Intel Daihen_mq \n");
    printf("daihen -c configfile.ini [-D] [-v] [-t] [-A]\n");
    printf("  where:\n");
    printf("  -c configfile.ini is mandatory config file\n");
    printf("  -D debug -> dumps everything\n");
    printf("  -v verbose -> gives status as processing is being undertaken\n");
    printf("  -A all -> all messages to queue even if power is "
        "lower than threshold\n");
    printf("\n");

    printf(" Primary Message Queue Format is:- \n"
        "\tS0,          (decimal 0.00 to 25.00)\n"
        "\tS1,          (decimal 0.00 to 25.00)\n"
        "\tS2,          (decimal 0.00 to 25.00)\n"
        "\tFoward_power, Forward_flag, (integer 0 to 2400), [integer 0:1]\n"
        "\tReflected_power, Reflected_flag, (integer 0 to 2400), [integer 0:1]\n"
        "\tUin, Uin_flag, (decimal -1.00 to +1.00), [integer 0:1]\n"
        "\tVin, Vin_flag, (decimal -1.00 to +1.00), [integer 0:1]\n"
        "\tUload, Uload_flag, (decimal -1.00 to +1.00), [integer 0:1]\n"
        "\tYload, Yload_flag, (decimal -1.00 to +1.00), [integer 0:1]\n");
    printf("\n");

    exit(0);
}

```

```

}

//-----

// Calc checksum to insert in 'three digit [000 to 255] checksum and ':'
int calc_tail(char * mq_bf, int mlen) {
int i;
int sum = 0;

for (i=0; i<mlen-4; i++) { // exclude already add 'digits' and ':'
    sum += mq_bf[i];
}
sum &= 0xff;
sprintf(&mq_bf[mlen-4], "%03d:", sum); // left justified 3 digits decimal

return 0;
}

//-----

int build_shaper_mq_header(char *buf,
                           int buf_len,
                           char *sensor_name,
                           char *data_type,
                           char *data_structure,
                           int elements,
                           char *separator,
                           int msg_len) {

int res;
char tim_buf[SM_BUF];

res = wr3_date(tim_buf, SM_BUF, DEC_4_PLACE);
if (res) {
    fprintf(stderr, "Error in datetime function\n");
}

res = snprintf(buf, buf_len, "%s;%s;%s;%s;%d;%s;%d",
               sensor_name,
               tim_buf,
               data_type,
               data_structure,
               elements,
               separator,
               msg_len);

if (buf_len < res) {
    fprintf(stderr, "Could not pack header, buf too small %d:%d\n", buf_len, res);
    return -1;
}
return res;
}

//-----

int is_number (char *s) {
int res = 0;
for (; s && *s; s++) {
    res = 1;
    if ((*s < '0') || (*s > '9'))
        return (0);
}
return (res);
}

//-----

int load_ini_parms(char * ini_file) {
char atunit[SM_BUF];
char client_id[24]; // temp

```

```

const char *cp;
dictionary *d = NULL;
struct hostent *hp;
int octet;
int host;
int j;

d = iniparser_load(ini_file);
if (NULL == d) {
    fprintf(stderr, "Could not load %s INI file !!\n", ini_file);
    exit(1);
}

// Autotuner stuff

snprintf(atunit, SM_BUF, "%s:%s", "AUTOTUNER", "ratio");
VDPRINT(("Read --> %s\n", atunit));
ratio = iniparser_getint(d, atunit, DEFAULT_RATIO);
if (ratio < 1) ratio = 1;
if (ratio > 15) ratio = 15;

snprintf(atunit, SM_BUF, "%s:%s", "AUTOTUNER", "powerthreshold");
VDPRINT(("Read --> %s\n", atunit));
forward_power_threshold = iniparser_getint(d, atunit, POWER_THRESHOLD);
if (forward_power_threshold < 1) forward_power_threshold = 1;
if (forward_power_threshold > 1000) forward_power_threshold = 1000;

snprintf(atunit, SM_BUF, "%s:%s", "AUTOTUNER", "device");
VDPRINT(("Read --> %s\n", atunit));
cp = iniparser_getstring(d, atunit, "NoDevice");
snprintf(devname, SM_BUF, "%s", cp);

snprintf(atunit, SM_BUF, "%s:%s", "AUTOTUNER", "sensorname");
VDPRINT(("Read --> %s\n", atunit));
cp = iniparser_getstring(d, atunit, "AT0");
snprintf(sensorname, SM_BUF, "%s", cp);

// Mosquitto Stuff

snprintf(atunit, SM_BUF, "%s:%s", "MOSQUITTO", "topicname");
VDPRINT(("Read --> %s\n", atunit));
cp = iniparser_getstring(d, atunit, "AT0");
snprintf(topicname, SM_BUF, "%s", cp);

// set up a reasonable random client id (can pass NULL to procedure!)
memset(client_id, 0, 24); // generate a temporary one using PID
snprintf(client_id, 23, "autotuner_%d", getpid());

snprintf(atunit, SM_BUF, "%s:%s", "MOSQUITTO", "clientid");
VDPRINT(("Read --> %s\n", atunit));
cp = iniparser_getstring(d, atunit, client_id);
snprintf(clientid, 23, "%s", cp); // only 24 chars for client id (inc NULL)

snprintf(atunit, SM_BUF, "%s:%s", "MOSQUITTO", "broker");
VDPRINT(("Read --> %s\n", atunit));
cp = iniparser_getstring(d, atunit, "localhost");
snprintf(sbroker, SM_BUF, "%s", cp);
// check the broker, and see if addressable
hp = gethostbyname(cp);
if (! hp) {
    fprintf(stderr, "Unknown host \"%s\"\n", cp);
    exit(1);
}
host = 0;
for (j=0; j<4; j++) {
    octet = hp->h_addr[j] & 0xff;
    //printf("%d\n", octet);
    host = (host << 8) | octet;
}
broker = host;

```



```

// VPRINT("host: %08x\n", host);

snprintf(atunit,SM_BUF,"%s:%s", "MOSQUITTO", "mqport");
VDPRINT(("Read --> %s\n", atunit));
mqport = iniparser_getint(d, atunit, MQTT_PORT);

snprintf(atunit,SM_BUF,"%s:%s", "MOSQUITTO", "keepalive");
VDPRINT(("Read --> %s\n", atunit));
keepalive = iniparser_getint(d, atunit, KEEPALIVE);

snprintf(atunit,SM_BUF,"%s:%s", "MOSQUITTO", "persist");
VDPRINT(("Read --> %s\n", atunit));
persist = iniparser_getboolean(d, atunit, FALSE);

iniparser_freedict(d);
d = NULL;

return 0;
}

//-----

void flags_init (int argc, char *argv[], int mode) {
    int i;
    for (i=1; i<argc; i++) {
        if (! strcmp(argv[i], "-v")) { flag_v = 1; continue; }
        if (! strcmp(argv[i], "-t")) { flag_t = 1; continue; }
        if (! strcmp(argv[i], "-D")) { flag_D = 1; continue; }
        if (! strcmp(argv[i], "-A")) { flag_A = 1; continue; }
        if (! strcmp(argv[i], "-c") && (i+1 < argc)) {
            flag_i = 1; arg_inifile = argv[i+1]; continue;
        }
    }

    fprintf(stderr, "unexpected flag \"%s\"\n", argv[i]);
    usage();
}

if (flag_i && arg_inifile) {
    if (strlen(arg_inifile) < sizeof(inifile)) {
        sprintf(inifile,"%s", arg_inifile);
    }
}

if (!flag_i) {
    fprintf(stderr, "You must specify an INI file\n");
    usage(); // does not return
}

load_ini_parms(inifile); // read some ini parms

printf("Inifile is      --> %s\n", inifile);
printf("Broker is       --> %d.%d.%d.%d : %d\n\r",
        (broker >> 24) & 0xff,
        (broker >> 16) & 0xff,
        (broker >> 8) & 0xff,
        (broker >> 0) & 0xff, mqport);
printf("ClientID is      --> %s\n", clientid);
printf("Topic is         --> %s\n", topicname);
printf("Sensor is        --> %s\n", sensorname);
printf("Device is         --> %s\n", devname);
printf("Ratio is          --> %d\n", ratio);
printf("Power Threshold is --> %d Watts\n", forward_power_threshold);
printf("Debug is          --> %d\n", flag_D);
printf("All Messages       --> %d\n", flag_A);
printf("Verbose is         --> %d\n", flag_v);
printf("Test Mode is       --> %d\n", flag_t);
printf("Acquisition Mode  --> %s\n", mode?"Complex":"Simple");
if (flag_H) {
    printf("Delimiter          --> %s\n", delimiter=="CR?"<CR> ':' on display":
        delimiter=="LF?"<LF>":
        delimiter=="NUL?"<NUL>":

```

```

        "Illegal");
printf("Baud is      --> %s\n", baud_rate==B9600?"9600":
        baud_rate==B4800?"4800":
        baud_rate==B2400?"2400":
        baud_rate==B1200?"1200":
        baud_rate==B600?"600":
        baud_rate==B300?"300":
        "Illegal");

printf("Data bits is  --> %s\n", data_bits==CS8?"8 bits":
        data_bits==CS7?"7 bits":
        data_bits==CS6?"6 bits":
        data_bits==CS5?"5 bits":
        "Illegal");

printf("Parity is     --> %s\n", parity_en==PARENB?"Enabled":
        "None");

if (parity_en) {
    printf("Parity type is  --> %s\n", parity==PARODD?"Odd":
            "Even");
}

printf("Stop bit      --> %s\n", stop_bits==CSTOPB?"2 Stop":
        "1 Stop");

}
fflush(NULL);
}

//-----

int simple_loop(void) {
int res,mos_res;
int rc;
char mq_buf[MQ_BUF];
char header_buf[SM_BUF];
char data_buf[MQ_BUF];
char time_buf[SM_BUF];
char *user_ptr = NULL;
int dl; // data length calculation
int ml; // message length

struct mosquitto *mosq;

    mosquitto_lib_init();

    mosq = mosquitto_new(clientid, true, user_ptr);
    if (mosq) {
        mosquitto_connect_callback_set(mosq, connect_callback);
        mosquitto_message_callback_set(mosq, message_callback);
        rc = mosquitto_connect(mosq, sbroker, mqport, keepalive);
        if (rc) {
            printf("%s Link DOWN %d\n",sensorname,rc);
            link_status = LINK_DOWN;
        }
        else {
            printf("%s Link UP\n",sensorname);
            link_status = LINK_UP;
        }
    }
    // main loop
    res = 0;
    while (keepRunning) {

// If you want to run without the actual machine connected ... !

#ifdef 1
        // pthread_mutex_lock(&timesync); // wait for sync (at start or end)
        if (!keepRunning) continue;

        res = process_msg(MSG_S0); // read S0

```

```

if(res) fprintf(stderr,"Error processing Stub 0\n");
res = process_msg(MSG_S1); // read S1
if(res) fprintf(stderr,"Error processing Stub 1\n");
res = process_msg(MSG_S2); // read S2
if(res) fprintf(stderr,"Error processing Stub 2\n");

// Now check if the Magnetron is ON/OFF, by reading the forward power
// and other salient values.
res = process_msg(MSG_MF); // read MF
if(res) fprintf(stderr,"Error processing Forward Power\n");
res = process_msg(MSG_MR); // reverse power
if(res) fprintf(stderr,"Error processing Reverse Power\n");
res = process_msg(MSG_MU); // input_reflection_coefficient_uin
if(res) fprintf(stderr,"Error processing UIN\n");
res = process_msg(MSG_MV); // input_reflection_coefficient_vin
if(res) fprintf(stderr,"Error processing VIN\n");
res = process_msg(MSG_LU); // load_reflection_coefficient_oload
if(res) fprintf(stderr,"Error processing ULOAD\n");
res = process_msg(MSG_LV); // load_reflection_coefficient_yload
if(res) fprintf(stderr,"Error processing YLOAD\n");

#endif

if (flag_t) // for debug on an autotuner not connected to a magnetron
    emulate_power();

#if 1

if ((forward_wave_power >= forward_power_threshold) || flag_A) {
    dl = snprintf(data_buf, MQ_BUF, DATA_FORMAT,
        stub_manual_setting[0], /* S0 */
        stub_manual_setting[1], /* S1 */
        stub_manual_setting[2], /* S2 */
        forward_wave_power, /* Forward Wave Power */
        (flag_fresh & FL_MF) ? 1:0, /* Fwr_flag */
        reflected_wave_power, /* Reflected Power */
        (flag_fresh & FL_MR) ? 1:0, /* Rev_flag */
        input_reflection_coefficient_uin, /* Input Coeff U */
        (flag_fresh & FL_MU) ? 1:0, /* Uin_flag */
        input_reflection_coefficient_vin, /* Input Coeff V */
        (flag_fresh & FL_MV) ? 1:0, /* Vin_flag */
        load_reflection_coefficient_oload, /* Uload_coeff */
        (flag_fresh & FL_LU) ? 1:0, /* Uload_flag */
        load_reflection_coefficient_yload, /* Yload_coeff */
        (flag_fresh & FL_LV) ? 1:0); /* Yload_flag */

    build_shaper_mq_header(header_buf,
        SM_BUF,
        sensorname,
        "CSV",
        "float",
        15,
        "\",\",",
        dl);

    ml = snprintf(mq_buf, MQ_BUF, "%s|s|s|03d|", header_buf,data_buf,0xff);
    calc_tail(mq_buf, ml);

#else

if ((forward_wave_power >= forward_power_threshold) || flag_A) {
    snprintf(mq_buf, MQ_BUF, MQ_STR,
        mymq_queue, /* Name */
        res?"1970-01-01 00:00:00.000":time_buf, /* Time */
        stub_manual_setting[0], /* S0 */
        stub_manual_setting[1], /* S1 */
        stub_manual_setting[2], /* S2 */
        forward_wave_power, /* Forward Wave Power */

```

```

(flag_fresh & FL_MF) ? 1:0,      /* Fwrд_flag */
reflected_wave_power,          /* Reflected Power */
(flag_fresh & FL_MR) ? 1:0,      /* Rev_flag */
input_reflection_coefficient_uin, /* Input Coeff U */
(flag_fresh & FL_MU) ? 1:0,      /* Uin_flag */
input_reflection_coefficient_vin, /* Input Coeff V */
(flag_fresh & FL_MV) ? 1:0,      /* Vin_flag */
load_reflection_coefficient_oload, /* Uload_coeff */
(flag_fresh & FL_LU) ? 1:0,      /* Uload_flag */
load_reflection_coefficient_yload, /* Yload_coeff */
(flag_fresh & FL_LV) ? 1:0);      /* Yload_flag */
#endif

switch (link_status) {
case LINK_UP:
    mos_res = mosquitto_publish(mosq, NULL, topicname, strlen(mq_buf),mq_buf, 0,false);
    if (mos_res) { // not good !
        link_status = LINK_DOWN;
        printf ("%s Link DOWN \n",sensorname);
    }
    break;
case LINK_DOWN:
    mos_res = mosquitto_reconnect(mosq); // attempt reconnect if broker went down
    if (!mos_res) {
        link_status = LINK_UP;
        printf ("%s Link UP \n",sensorname);
    }
    break;
default:
    printf ("%s Internal error on link status\n",sensorname);
    break;
}
}
else {
    ;; // Low power AND Suppress
}

if ((!res) && flag_D) {
    printf ("%s Time:%lld,FR:%x,S0:%.2f,S1:%.2f,S2:%.2f"
        ",last power:%d[%d],%d[%d],%f[%d],%f[%d],%f[%d],%f[%d]\n" ,
        sensorname,
        current_timestamp(),flag_fresh,
        stub_manual_setting[0],
        stub_manual_setting[1],
        stub_manual_setting[2],
        forward_wave_power,
        (flag_fresh & FL_MF) ? 1:0,
        reflected_wave_power,
        (flag_fresh & FL_MR) ? 1:0,
        input_reflection_coefficient_uin,
        (flag_fresh & FL_MU) ? 1:0,
        input_reflection_coefficient_vin,
        (flag_fresh & FL_MV) ? 1:0,
        load_reflection_coefficient_oload,
        (flag_fresh & FL_LU) ? 1:0,
        load_reflection_coefficient_yload,
        (flag_fresh & FL_LV) ? 1:0);
}
// since we have published these .. we will now mark as stale
flag_fresh &= ~(FL_S0 | FL_S1 | FL_S2 |
    FL_MF | FL_MR |
    FL_MU | FL_MV |
    FL_LU | FL_LV ); // add as appropriate here

pthread_mutex_lock(&timesync); // wait for synchronization
res = tstring(time_buf, SM_BUF); // get the global time, rather than when
    // we have sent/processed lots of
    // serial stuff (~10s of mS)
} // End (while(keeprunning))

// fall out and tidy up !

printf ("Res returned -> %d\n",res);

```

```

    mosquito_disconnect(mosq);
    mosquito_destroy(mosq);
    printf("Disconnected from broker\n");
    fflush(NULL);

    return EXIT_SUCCESS;
}

//-----

int main(int argc, char *argv[]) {
    int res;
    struct itimerval timer;

    /* Initial timeout value */
    timer.it_value.tv_sec = 1;          // initial timer is 1.5 seconds
    timer.it_value.tv_usec = 500000;

    timer.it_interval.tv_sec = 0;       // initial interval is 0.5 seconds
    timer.it_interval.tv_usec = 500000;

    // Register Signal Handler
    signal(SIGALRM, &sigalrm_handler);
    // register periodic timer with kernel
    setitimer(ITIMER_REAL, &timer, NULL);

    if (pthread_mutex_init(&timesync, NULL) != 0) {
        fprintf(stderr, "mutex init failed\n");
        return -1;
    }

    res = check_devices();
    signal(SIGINT, &intHandler); // assign signal handler
    signal(SIGTERM, &intHandler);

    flags_init(argc, argv, 0);

    res = simple_loop();

    return(res);
}

```

Appendix O Recording of Daihen Measurements

This is a sample run recording of the Daihen measurements as the Hitachi 712 is processing a recipe.

time	forward	s1	s2	s3	uin	uload	Vload
10202.3	5	0	0	0	0	0	0
10202.75	4	0	0	0	0	0	0.001
10203.2	498	0	0	0	0.058	0.073	0.031
10203.62	497	0.25	8.9	10.3	-0.014	-0.011	0.039
10204.1	498	0.25	8.9	10.3	-0.004	-0.016	0.043
10204.57	498	0.25	8.9	10.3	0.002	-0.018	0.048
10205.03	497	0.25	8.9	10.3	0.014	-0.018	0.053
10205.45	498	0.25	8.9	10.3	0.012	-0.018	0.054
10205.92	498	0.25	8.9	10.3	0.01	-0.018	0.054
10206.37	498	0.25	8.9	10.3	0.005	-0.018	0.053
10206.82	498	0.25	8.9	10.3	0.008	-0.022	0.052
10207.28	498	0.25	8.9	10.3	0.007	-0.016	0.051
10207.7	498	0.25	8.9	10.3	0.002	-0.02	0.05
10208.17	498	0.25	8.9	10.3	0.007	-0.018	0.049
10208.65	497	0.25	8.9	10.3	-0.028	-0.012	0.04
10209.1	498	0.25	8.9	10.3	0.014	-0.004	0.04
10209.55	497	0.25	8.9	10.3	0.027	0.012	0.045
10210	497	0.25	8.9	10.3	0.024	0.023	0.04
10210.42	498	0.25	8.9	10.3	0.027	0.024	0.034
10210.9	498	0.25	8.9	10.3	0.04	0.03	0.032
10211.33	498	0.25	8.9	10.3	0.035	0.028	0.031
10211.77	497	0.25	8.9	10.3	0.042	0.028	0.031
10212.25	497	0.25	8.9	10.35	0.037	0.031	0.031
10212.7	498	0.25	8.9	10.35	0.036	0.026	0.03
10213.17	498	0.25	8.9	10.35	0.032	0.035	0.03
10213.65	498	0.25	8.9	10.35	0.03	0.033	0.03
10214.12	498	0.25	8.9	10.35	0.03	0.028	0.029
10214.57	497	0.25	8.9	10.35	0.031	0.029	0.029
10215.03	498	0.25	8.9	10.35	0.026	0.03	0.029
10215.5	498	0.25	8.9	10.35	0.029	0.026	0.029
10215.95	498	0.25	8.9	10.35	0.031	0.026	0.029
10216.4	498	0.25	8.9	10.35	0.026	0.028	0.028
10216.87	497	0.25	8.9	10.35	0.028	0.026	0.028
10217.33	498	0.25	8.9	10.35	0.033	0.028	0.028
10217.77	498	0.25	8.9	10.35	0.027	0.026	0.028
10218.2	498	0.25	8.9	10.35	0.026	0.026	0.028
10218.67	498	0.25	8.9	10.35	0.028	0.024	0.028
10219.17	498	0.25	8.9	10.35	0.028	0.026	0.028
10219.62	498	0.25	8.9	10.35	0.034	0.026	0.028

10220.05	498	0.25	8.9	10.35	0.029	0.026	0.028
10220.45	498	0.25	8.9	10.35	0.028	0.024	0.028
10220.9	498	0.25	8.9	10.35	0.027	0.03	0.028
10221.4	498	0.25	8.9	10.35	0.028	0.031	0.027
10221.85	498	0.25	8.9	10.35	0.027	0.028	0.027
10222.33	498	0.25	8.9	10.35	0.029	0.024	0.027
10222.75	498	0.25	8.9	10.35	0.027	0.026	0.027
10223.2	498	0.25	8.9	10.35	0.027	0.024	0.027
10223.67	498	0.25	8.9	10.35	0.03	0.026	0.027
10224.15	498	0.25	8.9	10.35	0.025	0.026	0.027
10224.6	498	0.25	8.9	10.35	0.025	0.026	0.027
10225.05	498	0.25	8.9	10.35	0.028	0.03	0.027
10225.53	497	0.25	8.9	10.35	0.027	0.028	0.027
10225.97	498	0.25	8.9	10.35	0.028	0.03	0.027
10226.42	498	0.25	8.9	10.35	0.031	0.026	0.026
10226.9	498	0.25	8.9	10.35	0.025	0.026	0.027
10227.35	498	0.25	8.9	10.35	0.024	0.026	0.026
10227.8	498	0.25	8.9	10.35	0.03	0.028	0.027
10228.25	498	0.25	8.9	10.35	0.027	0.03	0.026
10228.7	498	0.25	8.9	10.35	0.027	0.028	0.026
10229.15	498	0.25	8.9	10.35	0.029	0.026	0.026
10229.6	498	0.25	8.9	10.35	0.026	0.03	0.026
10230.05	498	0.25	8.9	10.35	0.027	0.028	0.026
10230.5	498	0.25	8.9	10.35	0.026	0.03	0.026
10230.95	498	0.25	8.9	10.35	0.024	0.028	0.026
10231.4	498	0.25	8.9	10.35	0.026	0.032	0.025
10231.9	497	0.25	8.9	10.35	0.033	0.026	0.026
10232.32	498	0.25	8.9	10.35	0.025	0.029	0.026
10232.78	498	0.25	8.9	10.35	0.025	0.029	0.026
10233.25	498	0.25	8.9	10.35	0.03	0.03	0.025
10233.7	498	0.25	8.9	10.35	0.025	0.031	0.025
10234.17	498	0.25	8.9	10.35	0.027	0.026	0.025
10234.65	498	0.25	8.9	10.35	0.026	0.026	0.025
10235.15	498	0.25	8.9	10.35	0.024	0.031	0.024
10235.63	498	0.25	8.9	10.35	0.028	0.03	0.024
10236.1	498	0.25	8.9	10.35	0.026	0.026	0.024
10236.55	498	0.25	8.9	10.35	0.026	0.03	0.024
10237	498	0.25	8.9	10.35	0.026	0.031	0.024
10237.47	498	0.25	8.9	10.35	0.026	0.029	0.024
10237.95	498	0.25	8.9	10.35	0.027	0.024	0.024
10238.4	498	0.25	8.9	10.35	0.025	0.03	0.024
10238.87	498	0.25	8.9	10.35	0.026	0.029	0.024
10239.35	498	0.25	8.9	10.35	0.025	0.03	0.024
10239.83	498	0.25	8.9	10.35	0.022	0.026	0.024
10240.3	498	0.25	8.9	10.35	0.026	0.029	0.024
10240.7	498	0.25	8.9	10.35	0.022	0.031	0.024

10241.12	498	0.25	8.9	10.35	0.023	0.029	0.024
10241.57	498	0.25	8.9	10.35	0.025	0.031	0.024
10242.05	498	0.25	8.9	10.35	0.022	0.033	0.023
10242.53	498	0.25	8.9	10.35	0.024	0.03	0.023
10242.97	498	0.25	8.9	10.35	0.024	0.026	0.024
10243.45	498	0.25	8.9	10.35	0.022	0.028	0.024
10243.9	498	0.25	8.9	10.35	0.024	0.03	0.023
10244.37	498	0.25	8.9	10.35	0.027	0.033	0.023
10244.88	498	0.25	8.9	10.35	0.023	0.029	0.023
10245.35	498	0.25	8.9	10.35	0.026	0.028	0.023
10245.83	498	0.25	8.9	10.35	0.023	0.031	0.024
10246.3	498	0.25	8.9	10.35	0.025	0.029	0.023
10246.75	498	0.25	8.9	10.35	0.022	0.031	0.023
10247.2	498	0.25	8.9	10.35	0.023	0.031	0.023
10247.67	498	0.25	8.9	10.35	0.025	0.031	0.023
10248.15	498	0.25	8.9	10.35	0.017	0.033	0.023
10248.62	498	0.25	8.9	10.35	0.022	0.03	0.023
10249.1	498	0.25	8.9	10.35	0.022	0.031	0.023
10249.55	498	0.25	8.9	10.35	0.026	0.03	0.023
10250	498	0.25	8.9	10.35	0.024	0.024	0.024
10250.47	498	0.25	8.9	10.35	0.023	0.028	0.023
10250.95	498	0.25	8.9	10.35	0.023	0.031	0.024
10251.42	498	0.25	8.9	10.35	0.023	0.031	0.023
10251.9	498	0.25	8.9	10.35	0.023	0.029	0.023
10252.35	498	0.25	8.9	10.35	0.023	0.03	0.023
10252.85	498	0.25	8.9	10.35	0.023	0.028	0.023
10253.33	498	0.25	8.9	10.35	0.022	0.026	0.023
10253.8	498	0.25	8.9	10.35	0.024	0.031	0.023
10254.25	497	0.25	8.9	10.35	0.017	0.029	0.023
10254.72	498	0.25	8.9	10.35	0.026	0.033	0.023
10255.17	498	0.25	8.9	10.35	0.005	0.026	0.028
10255.65	298	0.2	9.15	10.6	0.042	-0.001	0.048
10256.1	298	0.15	9.15	10.7	0.053	0.039	0.038
10256.55	297	0.1	9.35	11.05	0.041	-0.013	0.049
10257.03	298	0.1	9.35	11.05	0.039	0.026	0.04
10257.47	297	0.1	9.4	11.1	0.022	0.021	0.028
10257.9	297	0.1	9.4	11.1	0.008	0.053	0.009
10258.3	297	0.1	9.05	11.1	-0.003	0.046	-0.011
10258.78	298	0.1	9	11.1	-0.016	0.046	-0.022
10259.2	298	0.1	9	11.1	-0.021	0.05	-0.027
10259.65	298	0.1	8.9	11.05	-0.021	0.039	-0.029
10260.12	297	0.1	8.9	11.05	-0.018	0.047	-0.03
10260.6	298	0.1	8.9	11.05	-0.018	0.043	-0.03
10261.05	297	0.1	8.9	11.05	-0.024	0.047	-0.03
10261.53	297	0.1	8.9	11.05	-0.016	0.047	-0.03
10262	297	0.1	8.9	11.05	-0.029	0.039	-0.03

10262.45	4	0	5.2	5.8	0	0	0
10262.9	0	0	0	0	0	0	0
10263.35	0	0	0	0	0	0	0
10263.8	0	0	0	0	0	0	0
10264.22	0	0	0	0	0	0	0

Appendix P Novellus Concept 2 DC Plasma Deposition Syetem and FRES

The Novellus Concept 2 was the first on-site test for the FRES acquisition system. The tool is a two-chamber system with a load-lock robot with each chamber having a six-position carousel. Six wafers are loaded onto the carousel and then the processing is performed. For proprietary reasons, the recipes are unavailable. Data views are for temporal data only on the primary 13.56 MHz RF drive signal. The machine in general is described as a high-density plasma reactor for gap-fill with the following features:-

- DLCM with Hi/Lo transfer capability for 150 mm and 200 mm wafers.
- Dual blade robot
- Integrated buffer station with unobstructed front access for manual loading of up to 4 cassettes
- Two speed-S dielectric CVD process module with electrostatic backside Helium cooling, source and bias RF power supplies.
- 4 channel ultraclean gas panel for Ar, O₂, SiH₄ and NF₃
- H₂ control
- High frequency RF generator running at ~1 kW
- Low frequency RF generator running at ~960 kHz
- Has support for SECS/GEM

A picture of the system is shown below with the attached FRES system and the additional OES system using an Ocean Optics USB2000plus (plasma is on in the picture).

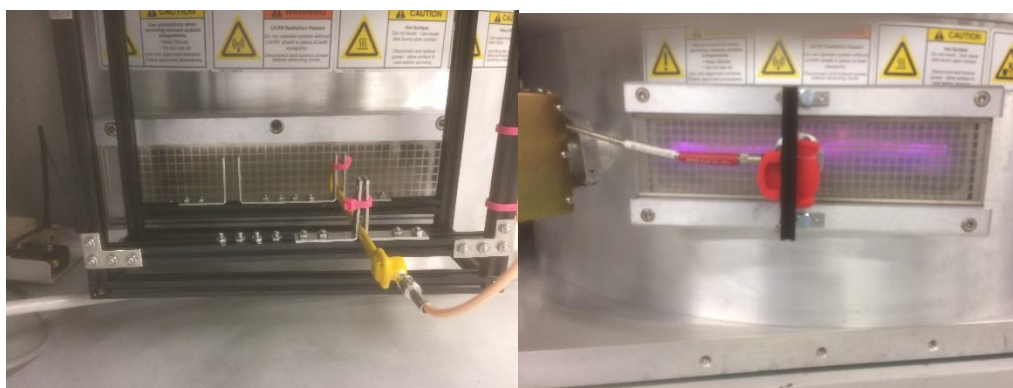


Figure 145 Novellus Concept 2 with FRES and OES

The signal for the FRES was very strong with a signal level of the maximum order of -25 dBm with the antenna located on the lower observation window below the sight line of the plasma. The OES was located coplanar with the plasma.

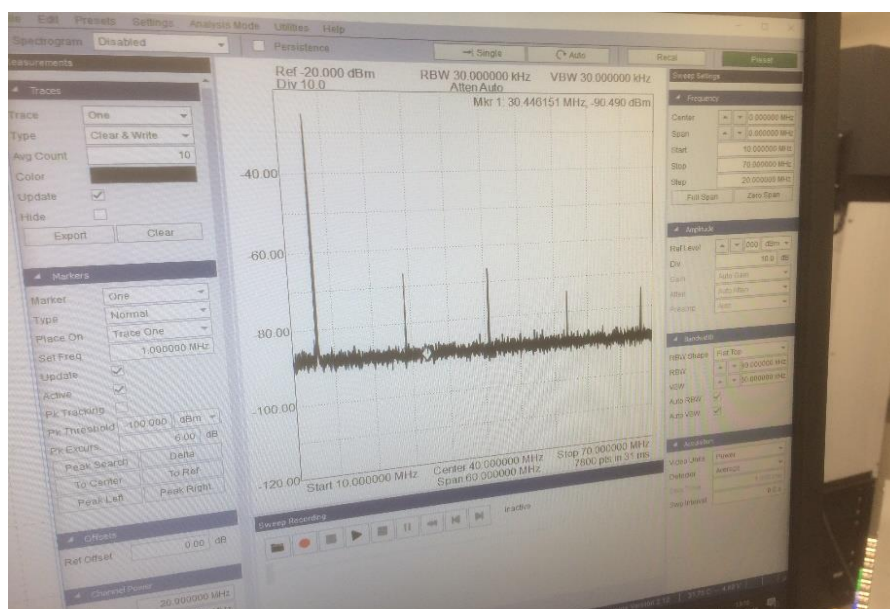


Figure 146 Novellus Concept 2 FRES reception

Due to clouding of the observation window and the use of a 'lossy' cable, the spectrum observed for the OES was of a very low level. This can be seen in the recorded spectrum with low levels of recorded emissions. The integration time was set to 'auto' for acquisition and even with a relatively large value, the signal was low.

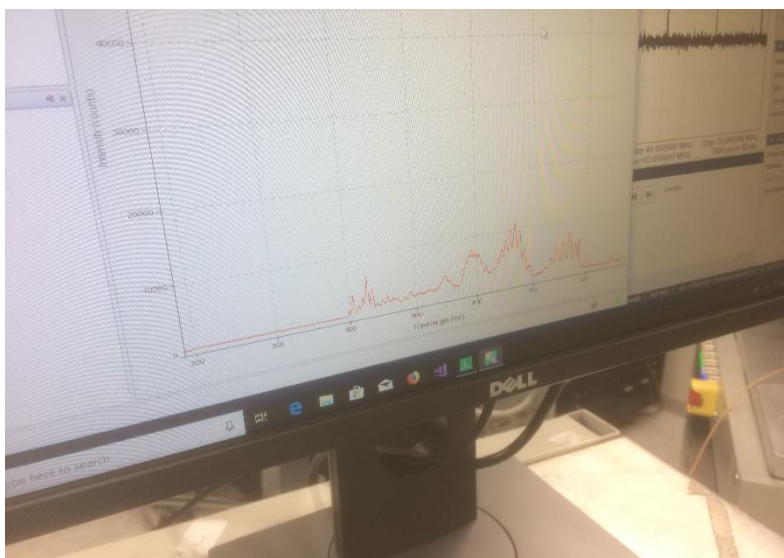


Figure 147 Novellus Concept 2 OES signal

The ADC in the OES spectrum analyzer gives values in the range 0 to 65K. The observed values were all under 10K, even with 'auto-shutter' in operation. Further work will be required such that significant species peaks are observable to match those with the FRES events.

Appendix Q Bespoke SDR system design for FRES and TRES

Q.1 Introduction

To acquire the required harmonics from 10 MHz to ~70 MHz and to provide high resolution with flexible design, the ADC subsystem driven by a FPGA was chosen. The Analog Devices AD9361 ADC combined with a programmable Phase Locked Loop (PLL) LO permits the sampling at 56 MHz over the 70 MHz to 6 GHz band. The basic design of the analyzer is a mixer with a PLL based LO followed by an amplified low pass filter (LPF) feeding into a high-speed ADC. This ADC operates a double data rate (DDR) 12bits parallel data bus at 112.8 MHz low voltage differential signaling) LVDS bus into the FPGA. The ADC and digital to analogue (DAC) sections have each 12bits of accuracy. This permits it collect at a continuous rate to permit 56 MHz bandwidth. An external mixer is used to up-convert the 1 MHz to 100 MHz in the design. This is to permit observation of frequencies down to 1 MHz. The block diagram is shown below (See Figure 148)

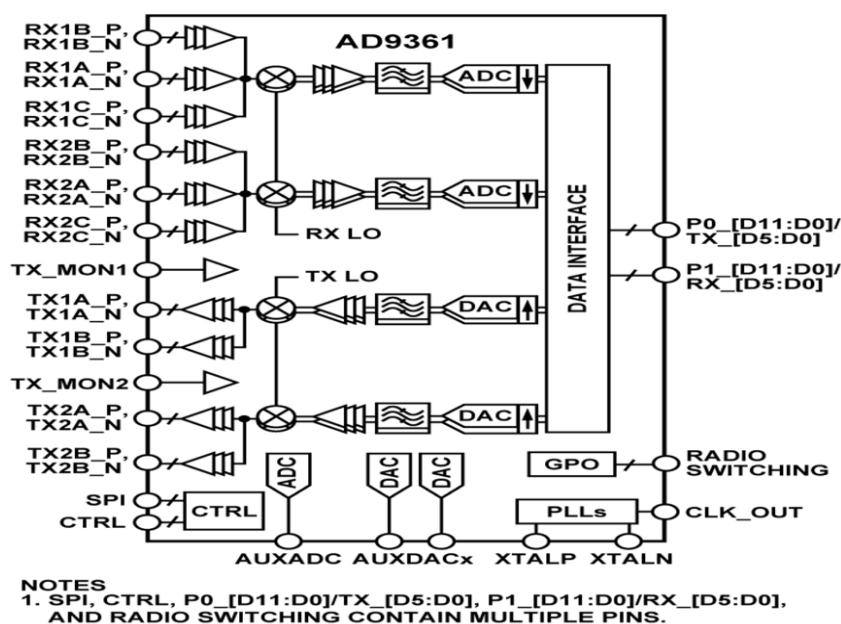


Figure 148 Block Diagram Mixer to ADC / DAC of Analog Devices AD9361

To permit the full acquisition range for the microwave and the capacitively-coupled plasmas, the initial design included a pre-mixer to up the frequency of

interest to the acquisition range of the AD9631 (70 MHz to 6 GHz). This allows the AD9631 to operate down to 1 MHz. The design utilizes an ADL5530 gain block followed by a 70 MHz LPF which feeds to the AD8342 mixer. The ADF4351 PLL LO is programmable but nominally set at 340 MHz. This is then brought out through a balun (balanced to unbalanced transformer) and finally a surface acoustic wave (SAW) bandpass filter (BPF) to feed into the AD9631 acquisition subsystem. The diagram is shown below (Figure 149).

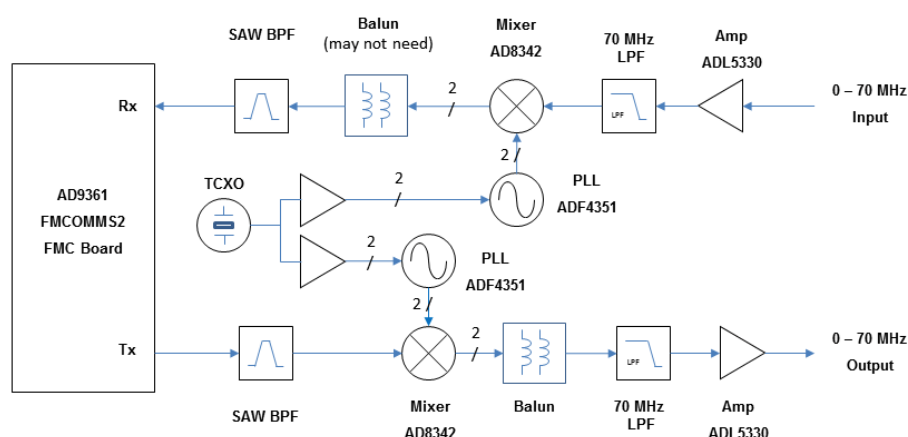


Figure 149 Block Diagram AD9631 SDR with Downconverter for Acquisition Down to 10 MHz

The physical PCB layout off broadband printed circuit boards (PCBs) is quite complex for the frequencies involved. Any vias in the PCB have a capacitance and appear as a disjoint in a circuit. The top layers of PCBs can be ordered with matching traces, such that the designer can specify microstrip sections of 100Ω to interconnect the analogue subsections. (The PCB manufacturer makes test boards and characterizes the PCB layout, the resins and the thickness of copper to obtain the required impedances. This permits the specification of the PCB design rules for the computer aided design (CAD) such that the engineer can layout the board accordingly). The layout of a SDR using the AD9631 and the down converter would be a very involved process. A test module for the AD9631 and the downconverter available from Analog Devices was purchased and mounted on a Xilinx Zynq-7000 FPGA board. The Zedboard is a test development board with a Xilinx Zynq® - 7000

System on a chip (SOC) with an ARM A9 processor and an uncommitted gate array section. The block diagram of the board is shown below (Figure 150):

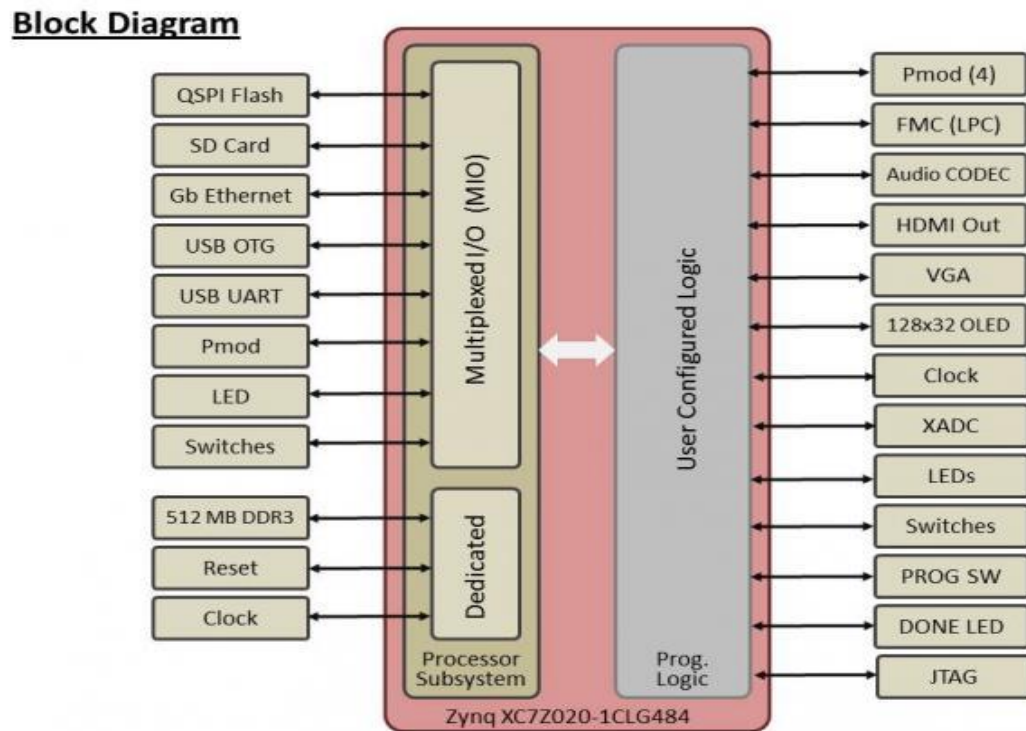


Figure 150 ZedBoard Block Diagram for SDR for Plasma Diagnostics

The board permits direct connection of the AD9361 based test module called the FM-COMMS3. A brief description of the FPGA mezzanine card (FMC) is described.

Q.1.1 FPGA Mezzanine Cards (FMC)

The FMC is a connector on Xilinx FPGA based development boards permitting the mounting of mezzanine cards to perform specific input and output functions. As we wanted to collect the parallel high-speed data from the AD9361 as it is performing its acquisition, this form factor permitted the ready interconnection between the FPGA and the ADC/DAC and serial peripheral interface (SPI) functions of the AD9361 and the SPI interface of the downconverter. The FMC comprises a low pin count connector (LPC) and high pin count (HPC) one. The FPGA input/out (IO) pins may be configured in low voltage differential signaling

(LVDS) mode with double data rate (DDR) to transfer the data between the AD9361 and the FPGA fabric. The ARM processor then has either a memory mapped memory section for data in and out of the AD9361 or a direct memory access (DMA) engine available to transfer the data to/from the ARM attached 512MB of DDR3 DRAM. The block diagram of the interconnected ARM processors, the FPGA fabric through the AXI interconnect allowing both DMA and memory mapping to take place is shown in the figure overleaf (Figure 151):

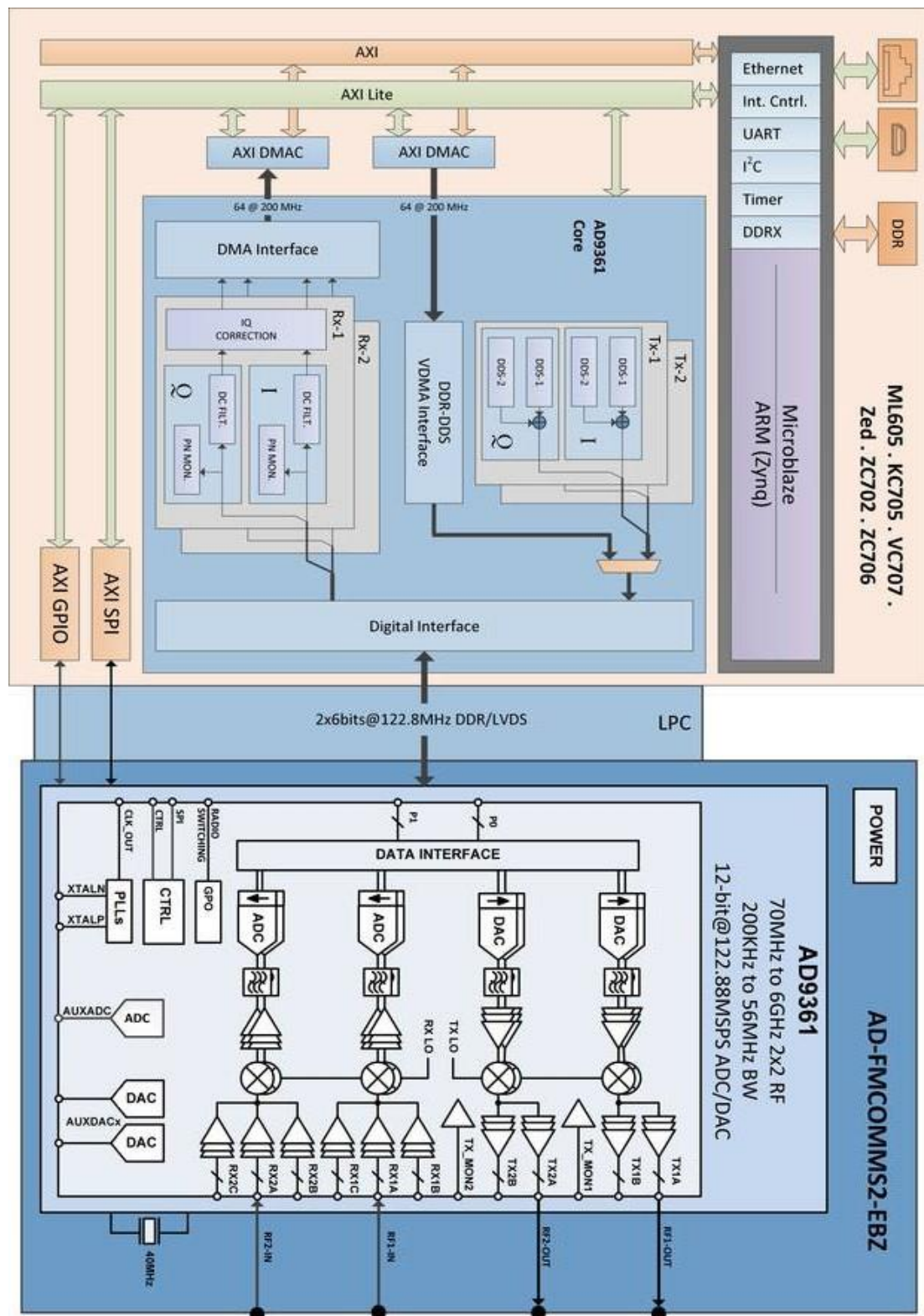


Figure 151 : Block Diagram of ARM Processors Through AXI to AD9361 SDR Design Using Zedboard

This then forms the basic SDR unit with the AD9361 performing the acquisition. Two channels are available for input and two are available for transmission. We

use the transmission channels to test the system by implementing a loopback. The Analog Devices frequency converter which down converts is based around a programmable LO and a mixer. It has both a RX section and a TX section and is designed to be mounted on the front connectors of the FMCOMMS3 module. The picture of this is shown below. It utilizes the power supply generated on the FMCOMMS3 and connects to the SPI interface JA1 on the Zedboard. The downconverter is shown in the figure below (Figure 152):

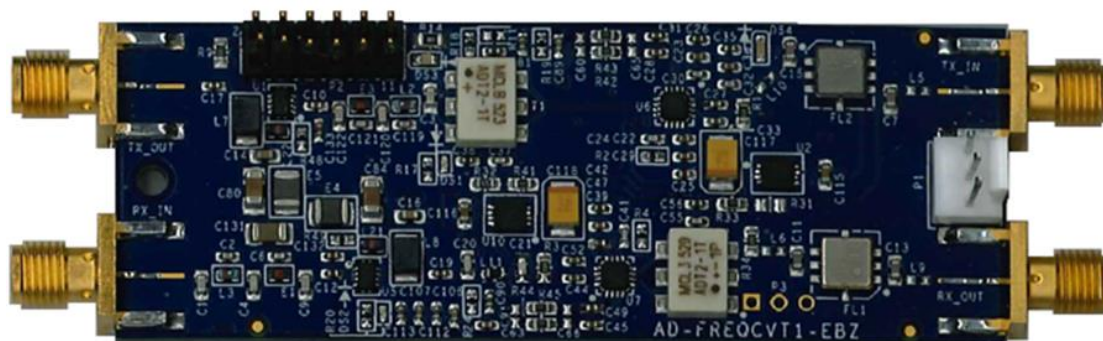


Figure 152 : Analog Devices Companion Downconverter FREQCVT1 for FMCOMMS3 / AD9361 Module

The spacing of the SMA connectors is identical to those of a single channel of the FMCOMMS3 module and permit direct connection using barrel SMA connectors.

Q.1.2 Operating the FMCOMMS3 / AD9361 / Downconverter Acquisition System

We created a SD card image on a 16GB SD card by generating the partitions required for the FPGA loader, the ARM configuration, the Linux bootstrap and the Linux image. This was built from various components supplied by Analog Devices to permit the testing of the FMCOMMS3 / AD9361 SDR module. Initially, the downconverter module FREQCVT1 was not connected and a loopback cable was installed between the RX and TX SMA connectors.

Q.1.3 Software Build for Analog Devices IIO Oscilloscope as Development Tool for SDR

The architecture of the system is to use a head computer with the analysis tools which can connect to a remote SDR spectrum analyzer over a network. We built

the libiio library to run on an Ubuntu 16.04 LTS system. We then built the latest (V 0.10) Analog Devices IIO Oscilloscope tool chain from sources to run on the Ubuntu host platform. The architecture is shown in the block diagram below:-

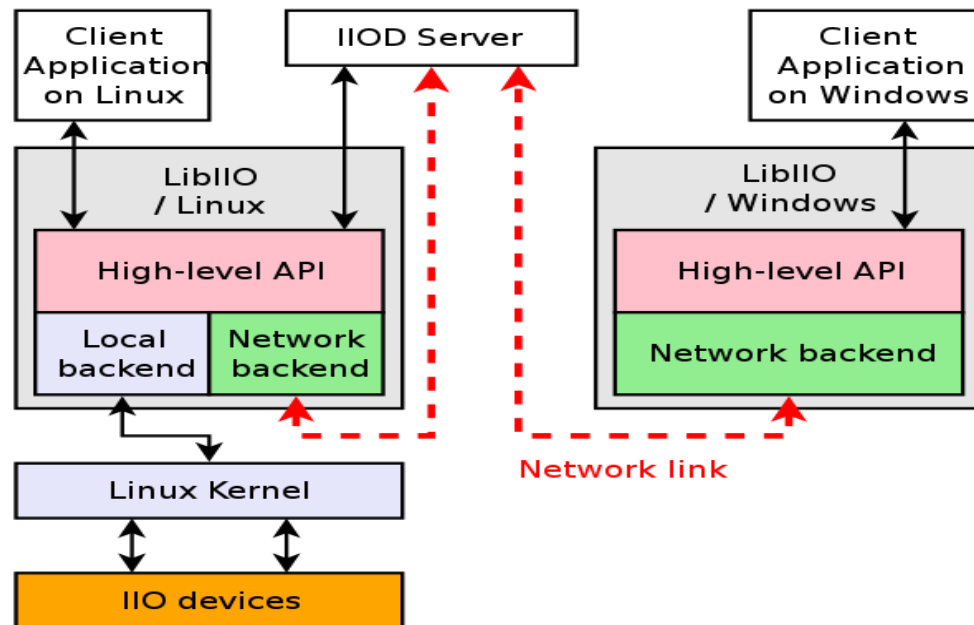


Figure 153 : Data Flow Rrchitecture of SDR Head with Remote Analysis

The IIO_Device which is the AD9361 and the associated FPGA interface are mapped through the Advanced eXtensible Interface (AXI) into the ARM address space. The Linux kernel on the Zedboard then implements the IIO driver to export the interfaces to the network or to be available to the local (Zedboard) machine. We then export the interfaces and the data ports of the AD9361 to the network and run a LibIIO client on the Ubuntu host to acquire the data and process it. The architecture was selected to complement the stated non-invasive sensor design goals, such that the SDR head could be located on the machine in the fab, but the data analysis would take place on a standard PC located remotely on the network. By utilizing the LibIIO components, the SDR head could be abstracted from the data storage and data analysis that could take place at a remote location from the machine and production floor. The IIO server in Figure 153 above runs on the Zedboard and the corresponding client runs on the remote Linux host. It is also possible to control the AD9361 and import/export the raw data from the SDR to MATLAB® using the LibIIO. GNURadio (GNURADIO.org, 2019) is another

option for displaying and analyzing the data stream from the IIOD server located on the Zedboard acquiring data from the AD9361. (Later we see the MATLAB analysis for the time domain analysis of the radio emission spectrum (TRES) for extracting the events that occur in the plasma). We then acquire the raw (in this case) data, I and Q data samples from the AD9361 and transfer then over the net to the host machine, where the fast Fourier transform (FFT) is performed and then the display module outputs this on the screen. For a production FRES device, we envisage performing the FFT directly on the FPGA and streaming the resultant frequency blocks back to the host processor for storage, display and mapping to plasma variations.

Q.1.4 Loopback testing of FMCOMMS3 / AD9361 and software displays

To test the configuration for basic operation, a SMA loopback coax was connected between the RX1 port and the TX1 port of the FMCOMMS3 / AD9361. The TX on channel 1 was configured to a LO of 2400.0 MHz with a RF bandwidth of 18.0 MHz and a sampling rate of 30.72 MHz for the I & Q samples with a single carrier wave (CW) tone of 0.999859 MHz. The RX channel 1 was configured for a LO of 2400.0 MHz with a similar bandwidth of 18.0 MHz and a sampling rate of 30.72MHz. The Finite Impulse Response (FIR) filter for the receiver was set to the LTE 20 MHz filter. LTE (4G mobile communication standard) specifies a real signal of 9 MHz passband but when upconverted occupies 18 MHz with an F_s of 30.72 MHz (i.e. +/- 9 MHz). The following figure (Figure 154) shows the captured 12 bit I & Q signal.

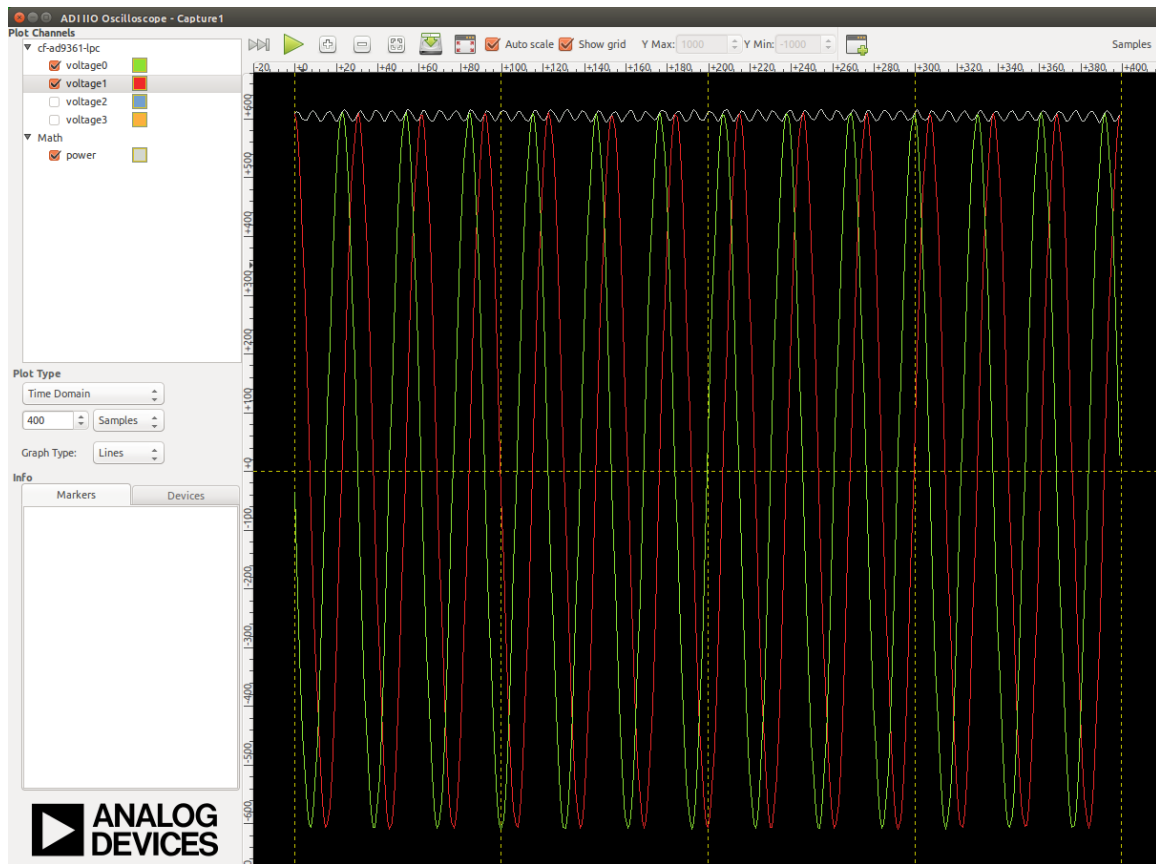


Figure 154 : TX-RX Loopback 0.999859MHz CW Tone with 12bit I and Q Raw Samples

The white line on the top of the sine wave in this Figure is the power calculated by the following equation:

$$\text{Power} = \sqrt{V_I^2 + V_Q^2}$$

where:

V_I = the voltage of the I sample

V_Q = the voltage of the Q sample

To confirm the linearity of the system, we plot a constellation diagram of the

V_I vs V_Q .

This should be a circle as the IQ samples for a sinewave describes a circle.

The following graph was obtained. (see Figure 155 below)

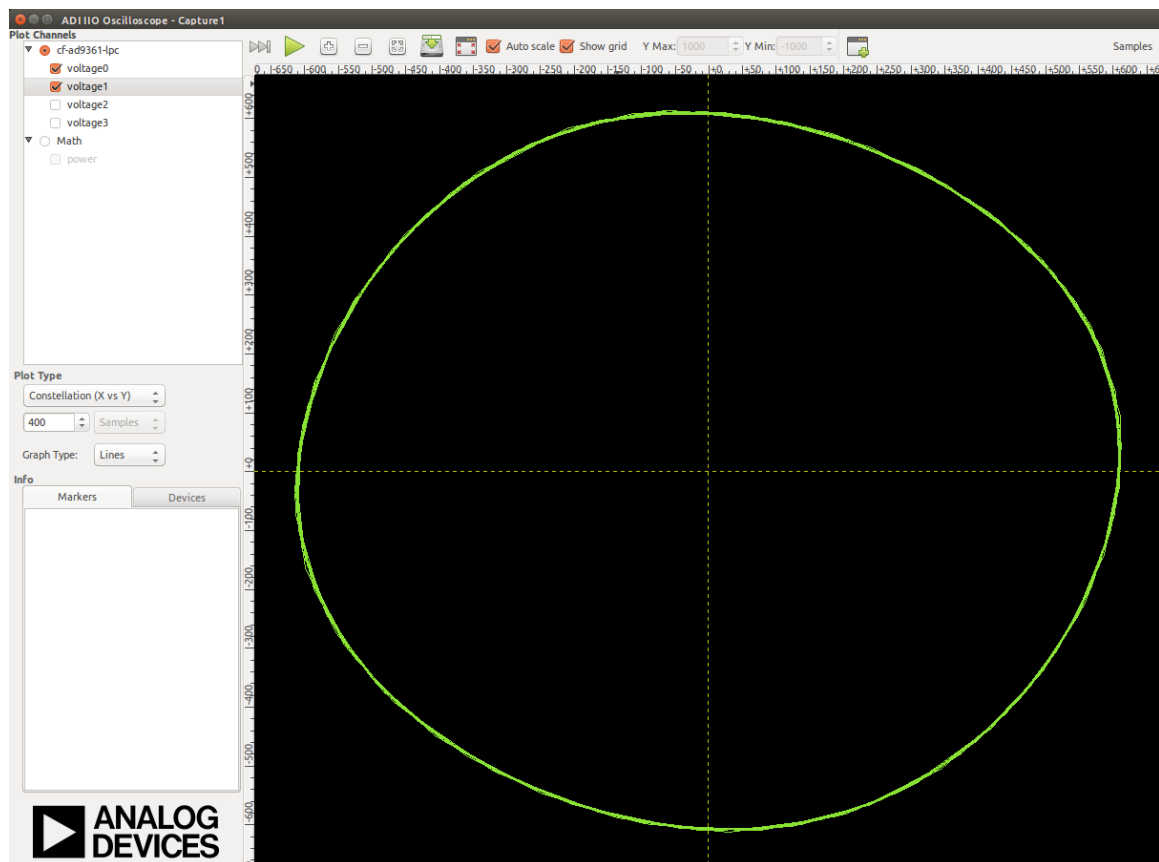


Figure 155: Constellation Plot of I & Q for CW of 0.999859 MHz (not accurate circle)

Note that the figure (Figure 155) shows a slight error in the circle. There are tuning values for the LO to compensate for crystal drift. There are also phase corrections that may be applied to the LO, but this should affect the TX and the RX equally. This seemed to be caused by clipping in the TX section as the CW tone amplitude was set to 0 dB of the full scale (dBFS). When the amplitude was set to -3 dBFS, the circle was round.

The following shows an example of the configuration parameters available for the frontend AD9361 acquisition subsystem.

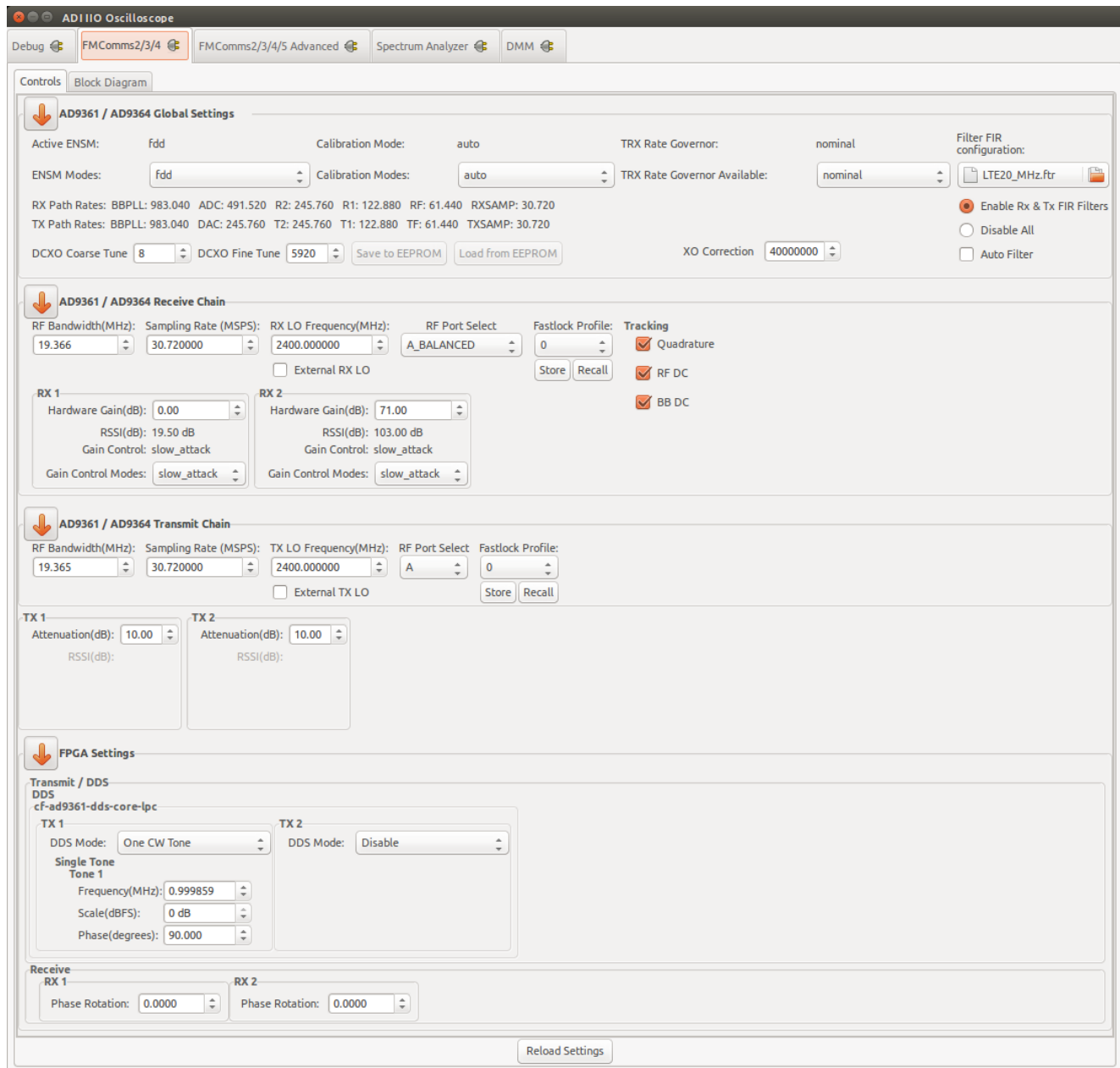


Figure 156 : Some of the Configuration Parameters Available for FMCOMMS3 / AD9361 TX/RX over SPI

It can be seen that a very wide range of radio parameters can be controlled through the SPI interface of the AD9361. This makes it a very suitable underlying acquisition platform for the FRES non-invasive sensor. For TRES, ZIF may permit more analysis to be performed on the transient nature of the radio spectrum acquired from the plasma. Below, we have a brief look at the parameters of a ZIF acquisition system based on a similar architecture to that outlined above.

Q.2 Time Domain Radio Emission Spectra, Redesign for Zero IF

The requirement for a TRES system as a non-invasive sensor dictates that high-speed events are continuously recorded. In the main, radio spectrum analyzers acquire a narrow-band spectrum band of data and store it for later FFT processing. To detect high speed events, the data must be continuously acquired. The design is for a flow-through zero IF acquisition for the base band frequency for the Oxford Instruments PlasmaLab 100 CCP system. Examining the results of the square-law, Hilbert or Wavelet transform, requires a pipelined architecture as the throughput rate is too high to process in software in a batch mode and samples would be missed that would permit events to go undetected.

For the TRES, using the same Zedboard, and software architecture, the FMCOMMS3 module could be replaced by the Analog Devices AD-FMCADC7-EBZ module which can sample from DC to 1.8GHz. The module features a 12bit ADC sampling at 2500 mega samples per second (MSPS) with a programmable front end 4.8 GHz differential amplifier. This design would be the basis for a high-end TRES analytics system.

Appendix R KNN Modelling Software

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% KNN Oxford Modelling // try and locate pressure regimes to
% correctly determine salient harmonics for gas-flow prediction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc;
clear all;
close all;

% The first row is the nomenclature
% The first column is the record number / time
% The second column is the Pressure (millitorr)
% The third column is the O2 scms
% the fourth column is the 'classification label'
% So extract the matrix
% The 'sheets' have the calculated 'averages' for the hyperspace

% Read an xlsx file
FilenameHarmonics_20 =
'Oxford_model\Data\Pressure_20_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_30 =
'Oxford_model\Data\Pressure_30_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_40 =
'Oxford_model\Data\Pressure_40_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_50 =
'Oxford_model\Data\Pressure_50_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';

% Pick up the 3 X B-Field Averaged array of harmonics
% start at 2nd row, take columns (F (6) : DB (106))
% (top left is 0,0) (101 'harmonics')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 20 Milli
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_20;
opts = detectImportOptions(FilenameHarmonics);
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '20milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics20milli_30scm = array([20:22],:);
%
opts.Sheet = '20milli_25scm';
```

```

array = readmatrix(FilenameHarmonics,opts);
harmonics20milli_25scm = array([20:22],:);
%
opts.Sheet = '20milli_20scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics20milli_20scm = array([20:22],:);
%

%%%%%%%%%%
% 30 Milli
%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_30;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '30milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics30milli_30scm = array([20:22],:);
%
opts.Sheet = '30milli_25scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics30milli_25scm = array([20:22],:);
%
opts.Sheet = '30milli_20scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics30milli_20scm = array([20:22],:);
%

%%%%%%%%%%
% 40 milli
%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_40;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '40milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics40milli_30scm = array([20:22],:);
%
opts.Sheet = '40milli_25scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics40milli_25scm = array([20:22],:);
%
opts.Sheet = '40milli_20scm';
array = readmatrix(FilenameHarmonics,opts);

```

```

harmonics40milli_20scm = array([20:22],:);
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 50 milli
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_50;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '50milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics50milli_30scm = array([20:22],:);
%
opts.Sheet = '50milli_25scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics50milli_25scm = array([20:22],:);
%
opts.Sheet = '50milli_20scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics50milli_20scm = array([20:22],:);
%

% create a hyperspaces for a known pressure
hyperspace20 = [harmonics20milli_30scm; harmonics20milli_25scm;
harmonics20milli_20scm];
hyperspace30 = [harmonics30milli_30scm; harmonics30milli_25scm;
harmonics30milli_20scm];
hyperspace40 = [harmonics40milli_30scm; harmonics40milli_25scm;
harmonics40milli_20scm];
hyperspace50 = [harmonics50milli_30scm; harmonics50milli_25scm;
harmonics50milli_20scm];

tags20 = ['harmonics20milli_30scm_A'; 'harmonics20milli_30scm_B';
'harmonics20milli_30scm_C'; ...
'harmonics20milli_25scm_A'; 'harmonics20milli_25scm_B';
'harmonics20milli_25scm_C'; ...
'harmonics20milli_20scm_A'; 'harmonics20milli_20scm_B';
'harmonics20milli_20scm_C'];

tags30 = ['harmonics30milli_30scm_A'; 'harmonics30milli_30scm_B';
'harmonics30milli_30scm_C'; ...

```

```

        'harmonics30milli_25scm_A'; 'harmonics30milli_25scm_B';
'harmonics30milli_25scm_C'; ...
        'harmonics30milli_20scm_A'; 'harmonics30milli_20scm_B';
'harmonics30milli_20scm_C'];

tags40 = ['harmonics40milli_30scm_A'; 'harmonics40milli_30scm_B';
'harmonics40milli_30scm_C'; ...
        'harmonics40milli_25scm_A'; 'harmonics40milli_25scm_B';
'harmonics40milli_25scm_C'; ...
        'harmonics40milli_20scm_A'; 'harmonics40milli_20scm_B';
'harmonics40milli_20scm_C'];

tags50 = ['harmonics50milli_30scm_A'; 'harmonics50milli_30scm_B';
'harmonics50milli_30scm_C'; ...
        'harmonics50milli_25scm_A'; 'harmonics50milli_25scm_B';
'harmonics50milli_25scm_C'; ...
        'harmonics50milli_20scm_A'; 'harmonics50milli_20scm_B';
'harmonics50milli_20scm_C'];

% We will not use auto Standardization, we will bound the values
% ourselves
% We know (?) the dBm values will be between +40dBm and -200dBm
% Actually from SA44, these are +20 and -120 (but leave some room)
% .. so ... normalize based on that
% val = (xi - min) / (max - min)
% Lets set boundaries
%
radiomin = -200.0;
radiomax = 40.0;

% copy for manipulation
hn20 = hyperspace20;
hn20 = (hn20 - radiomin) ./ (radiomax - radiomin);

hn30 = hyperspace30;
hn30 = (hn30 - radiomin) ./ (radiomax - radiomin);

hn40 = hyperspace40;
hn40 = (hn40 - radiomin) ./ (radiomax - radiomin);

hn50 = hyperspace50;
hn50 = (hn50 - radiomin) ./ (radiomax - radiomin);

```

```

%% Just to plot in 3D so as to be able to view from different
perspectives
% plot3([1:101],[1:101],hn);
% title('B-Field Averages Antenna measurement for change in flow rates
and pressure');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('Normalized Tagged B Field');
% plot([1:101],hn);

% [coeff20, score20, latent20, tsd20, variance20 ] = pca(hn20);
% title('B-Field PCA Averages Antenna measurement for change in flow
rates and pressure = 20millitorr');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('Principal Tagged B Field');
%
% [coeff30, score30, latent30, tsd30, variance30 ] = pca(hn30);
% title('B-Field PCA Averages Antenna measurement for change in flow
rates and pressure = 30millitorr');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('Principal Tagged B Field');
%
% [coeff40, score40, latent40, tsd40, variance40 ] = pca(hn40);
% title('B-Field PCA Averages Antenna measurement for change in flow
rates and pressure = 40millitorr');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('Principal Tagged B Field');
%
% [coeff50, score50, latent50, tsd50, variance50 ] = pca(hn50);
% title('B-Field PCA Averages Antenna measurement for change in flow
rates and pressure = 50millitorr');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('Principal Tagged B Field');

%%
%%
% %plot([1:101],coeff(:,1:4));
% hold on
% plot([1:20],coeff20(:,1), 'r');
% plot([1:20],coeff30(:,1), 'g');
% plot([1:20],coeff40(:,1), 'm');
% plot([1:20],coeff50(:,1), 'b');
% legend('PC1 for 20millitorr','PC1 for 30millitorr','PC1 for
40millitorr','PC1 for 50millitorr');
% title('Plot of first Coeff mappings for B-Field PCA Averages Antenna
measurement for change in flow rates and pressure of
20,30,40,50millitorr');
% xlabel('Harmonic Number (of 13.5596MHz)');
% ylabel('PCA component loadings');

```

```

% hold off;
% %

%
% for p = 20:50:10
%
%     for c = 1:5
%         fprintf('The %d basis vector accounts for %.5f%% of the
variance at %d millitorr \n',c,variance20(c,1),p)
%     end
% end
%
% plot([1:2],variance20);
% title('B-Field Variance of Averages Antenna measurement for change in
flow rates and pressure of 20millitorr');
% xlabel('Principal Component Number');
% ylabel('Percentage coverage for basis vector');
%
% plot([1:2],variance30);
% title('B-Field Variance of Averages Antenna measurement for change in
flow rates and pressure of 30millitorr');
% xlabel('Principal Component Number');
% ylabel('Percentage coverage for basis vector');
%
% plot([1:2],variance40);
% title('B-Field Variance of Averages Antenna measurement for change in
flow rates and pressure of 40millitorr');
% xlabel('Principal Component Number');
% ylabel('Percentage coverage for basis vector');
%
% plot([1:2],variance50);
% title('B-Field Variance of Averages Antenna measurement for change in
flow rates and pressure of 50millitorr');
% xlabel('Principal Component Number');
% ylabel('Percentage coverage for basis vector');

% Now for KNN

model20 = fitcknn(hn20, tags20, 'NumNeighbors',1);
model30 = fitcknn(hn30, tags30, 'NumNeighbors',1);
model40 = fitcknn(hn40, tags40, 'NumNeighbors',1);
model50 = fitcknn(hn50, tags50, 'NumNeighbors',1);

```

```

% model.NumNeighbors = 3;
% distance = resubLoss(model);
% %display distance
% fprintf('Distance for neighbours of %i :',model.NumNeighbors);
% disp(distance);
%
%
% model.NumNeighbors = 5;
% distance = resubLoss(model);
% %display distance
% fprintf('Distance for neighbours of %i :',model.NumNeighbors);
% disp(distance);
%
% model.NumNeighbors = 1;
% rng(1); % For reproducibility
% CVmodel = crossval(model);
% classError = kfoldLoss(CVmodel);
%
% disp('Cross validate, get classification error : ');
% disp(classError);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% So now read the Data and check prediction vs actual
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The data to be run through the model
% The first row is the nomenclature
% The first column is the record number / time
% The second column is the Pressure (millitorr)
% The third column is the O2 scms
% The fourth column is the RUN
% the fifth column is the 'classification label'

test_data_file = 'Oxford_model\Data\Pre_O2_Predict_ABC.xlsx';
opts = detectImportOptions(test_data_file);

%check the 20 millitorr first
opts.Sheet = 'Sheet1';
preview(test_data_file,opts)
opts.SelectedVariableNames = [6:106];

% Read the data
test_data_array = readmatrix(test_data_file,opts);

```

```

% pick up the pressure column to apply to the correct model
opts.SelectedVariableNames = [2];
pressure_data_array = readmatrix(test_data_file,opts);

% Pick up the labels
opts.SelectedVariableNames = [5];
labels = readmatrix(test_data_file,opts);


% Need to normalize the values ... as above
prdict_data_n = (test_data_array - radiomin) ./ (radiomax - radiomin);


% So now .. lets test the hypothesis, with single NN
model.NumNeighbors = 1;


%% Test
tmp = size(test_data_array); % get the length of the data sample to
test
loopcount = tmp(1);

fprintf("Elements -> %i\n", loopcount);

for c = 1:loopcount % iterate over the length

    my_row = c; % Pick the particular row

    % now extract (the normalized) row from that data
    test_data_row = prdict_data_n(my_row,:);

    %% Just for Grins .. how about 'overtesting' .. supply the raw data
    %% test_data_row = hn(my_row,:);

    % offer up to the Hyperspace
    if 20 == pressure_data_array(my_row)
        [oplabel,opscore,opcost] = predict(model20, test_data_row);
    elseif 30 == pressure_data_array(my_row)
        [oplabel,opscore,opcost] = predict(model30, test_data_row);
    elseif 40 == pressure_data_array(my_row)

```



```

        [oplabel,opscore,opcost] = predict(model40, test_data_row);
    elseif 50 == pressure_data_array(my_row)
        [oplabel,opscore,opcost] = predict(model50, test_data_row);
    else
        fprintf('Error in Pressure Column index
%d\n',pressure_data_array(my_row));
        return;
    end
    %
    fprintf("For Row %d at Pressure %d Known -> %s, predicted -> %s
\n", my_row, pressure_data_array(my_row), labels{my_row},oplabel);
    % disp (labels(my_row));
    %% Test
    %% disp (labels(my_row));
    % fprintf("Predicted ->");
    % disp(oplabel);
    % disp(opscore);
    % disp(opcost);
    % disp('-----');
end

return;

```

Appendix S Laplace Transform Modelling Software

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Laplace Oxford Modelling
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc;
clear all;
close all;

% Some constants for the system
% Drive frequency
f=13599600; % Hz
% Drive amplitude of the primary
AMP = 1;

% OP = TF . IN
% TF = OP / IN

syms s IN TF20 TF30 TF40 TF50 OUT20 OUT30 OUT40 OUT50;

% dimension an array for the measured frequencies
w = ones(1,103);

for c=1:103
    w(c) = c * 2 * pi * f;
end

% Basic sinewave drive frequency
% IN = sin(wt) - L -> w/(s^2 + w^2), where w = 2pif
% Generate the input function
IN = AMP * ((w(1)) / (s^2 + w(1)^2)) ;
```

```

% The first row is the nomenclature
% The first column is the record number / time
% The second column is the Pressure (millitorr)
% The third column is the O2 scms
% the fourth column is the 'classification label'
% So extract the matrix
% The 'sheets' have the calculated 'averages' for the hyperspace

% Read an xlsx file
FilenameHarmonics_20 =
'Oxford_model\Data\Pressure_20_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_30 =
'Oxford_model\Data\Pressure_30_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_40 =
'Oxford_model\Data\Pressure_40_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';
FilenameHarmonics_50 =
'Oxford_model\Data\Pressure_50_Pre_O2_Learn_20_25_30_scm_ABC.xlsx';

% Pick up the B-Field Averaged array of harmonics
% start at 2nd row, take columns (F (6) : DB (106))
% (top left is 0,0) (101 'harmonics')

%%%%%%%%%%
% 20 Milli
%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_20;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '20milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics20milli_30scm = array(19,:);

%%%%%%%%%%
% 30 Milli
%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_30;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages

```

```

opts.Sheet = '30milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics30milli_30scm = array(19,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 40 milli
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_40;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% Now pick the pre-calculated averages
opts.Sheet = '40milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics40milli_30scm = array(19,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 50 milli
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FilenameHarmonics = FilenameHarmonics_50;
opts = detectImportOptions(FilenameHarmonics);
%
preview(FilenameHarmonics,opts)
opts.SelectedVariableNames = [6:106];
%
% % Now pick the pre-calculated averages
opts.Sheet = '50milli_30scm';
array = readmatrix(FilenameHarmonics,opts);
harmonics50milli_30scm = array(19,:);
%

% .. Have four different pressures (50,40,30,20 millitorr)

har20 = harmonics20milli_30scm;
har30 = harmonics30milli_30scm;
har40 = harmonics40milli_30scm;
har50 = harmonics50milli_30scm;

```

```

% H1 and H2 were not recorded for the FRES Pressure and Gas-flow
% % experiments as they remain constant and flood
% radio due to their high levels. We will simulate
% these from antennae measurements made earlier

H1 = -22.5;
H2 = -31.0;

% We need to add these columns to the data array
% Create column the correct size and scale values appropriately
h1=ones(size(har20,1),1) .* H1;
h2=ones(size(har20,1),1) .* H2;

% Insert the columns

hn20 = [h1 h2 har20];
hn30 = [h1 h2 har30];
hn40 = [h1 h2 har40];
hn50 = [h1 h2 har50];

% The values recorded (leave log dBm) and change to linear values of
% power
% Convert to linear values (-dB is small number !)

hnlin20 = hn20 ./ 10;
hnlin30 = hn30 ./ 10;
hnlin40 = hn40 ./ 10;
hnlin50 = hn50 ./ 10;

% convert dB to linear value ( gain = 10^(db/10) .. power) ( gain =
10^(db/20) .. voltage)
hnlin20 = 10 .^ hnlin20;
hnlin30 = 10 .^ hnlin30;
hnlin40 = 10 .^ hnlin40;
hnlin50 = 10 .^ hnlin50;

% Get the frequencies squared
w2 = w.^2;

% Combine to create the equation
% Set the first symbol, will concatenate after this

```

```

OUT20 = hnlin20(1) * w(1) / (s^2 + w2(1));
OUT30 = hnlin30(1) * w(1) / (s^2 + w2(1));
OUT40 = hnlin40(1) * w(1) / (s^2 + w2(1));
OUT50 = hnlin50(1) * w(1) / (s^2 + w2(1));

% some way to concatenate these ??
for c=2:103;
    OUT20 = [OUT20 + (hnlin20(c) * (w(c) / (s^2 + w2(c))))];
    OUT30 = [OUT30 + (hnlin30(c) * (w(c) / (s^2 + w2(c))))];
    OUT40 = [OUT40 + (hnlin40(c) * (w(c) / (s^2 + w2(c))))];
    OUT50 = [OUT50 + (hnlin50(c) * (w(c) / (s^2 + w2(c))))];
end;

% Transfer function of the plasma
TF20 = OUT20 / IN ;
TF30 = OUT30 / IN ;
TF40 = OUT40 / IN ;
TF50 = OUT50 / IN ;

% Get the time domain transfer function
% By default, the independent variable is 's' and
% the transformation variable is 't'
f20 = ilaplace(TF20);
f30 = ilaplace(TF30);
f40 = ilaplace(TF40);
f50 = ilaplace(TF50);

% This will be in terms of Dirac and Heaviside (step function)
% plot the time function of the TRANSFER FUNCTION
% .. must convolve this with input

% Static plots
figure();
fplot (f20, [-50.0e-9 100.0e-9]) % for -50ns to +100ns either side
title('Inverse Laplace Time domain plot of Transfer function
(20millitorr)');
xlabel('nanoseconds');

```

```

ylabel('Voltage (power in receiver antenna)');

figure();
fplot (f30, [-50.0e-9 100.0e-9]) % for -50ns to +100ns either side
title('Inverse Laplace Time domain plot of Transfer function
(30millitorr)');
xlabel('nanoseconds');
ylabel('Voltage (power in receiver antenna)');

figure();
fplot (f40, [-50.0e-9 100.0e-9]) % for -50ns to +100ns either side
title('Inverse Laplace Time domain plot of Transfer function
(40millitorr)');
xlabel('nanoseconds');
ylabel('Voltage (power in receiver antenna)');

figure();
fplot (f50, [-50.0e-9 100.0e-9]) % for -50ns to +100ns either side
title('Inverse Laplace Time domain plot of Transfer function
(50millitorr)');
xlabel('nanoseconds');
ylabel('Voltage (power in receiver antenna)');

figure();
%fplot (f20, [-50.0e-9 100.0e-9]) % for -50ns to +100ns either side
h = fplot (f20, [-50.0e-9 100.0e-9]); % for -50ns to +100ns either side
xlabel('nanoseconds');
ylabel('Voltage (power in receiver antenna)');

% for i=1:100 for video capture
for i=1:100
    set (h,'function', f20);
    title('Inverse Laplace Time domain plot of Transfer function
(20millitorr)');
    drawnow
    pause(.5);

```

```

    set (h,'function', f30);
    title('Inverse Laplace Time domain plot of Transfer function
(30millitorr)');
    drawnow
    pause(.5);

    set (h,'function', f40);
    title('Inverse Laplace Time domain plot of Transfer function
(40millitorr)');
    drawnow
    pause(.5);

    set (h,'function', f50);
    title('Inverse Laplace Time domain plot of Transfer function
(50millitorr)');
    drawnow
    pause(.5);

end

% Lets find the roots
%sol20 = solve(TF20 == 0);
%sol30 = solve(TF30 == 0);
%sol40 = solve(TF40 == 0);
%sol50 = solve(TF50 == 0);

% Lets set a scale
% Real direction (-2 to 1 in 0.1 steps)
x=-4.5e9:1e8:4.5e9;
% Just to 'narrow it down in the real plane
% x=-2.5e9:1e8:2.5e9;
%x=-40:.25:40;

% Imaginary direction (-2 to 2 in 0.1 steps)
y=-4.5e9:1e8:4.5e9;
%y=-40:.25:40;

% create grid (automatically)
[X,Y]=meshgrid(x,y);

s=(X+1i*Y);

```



```
% Evaluate over the meshgrid of (alpha + omega i)
% The Omega part infers the Sinusoid parts and the Alpha
% represents the Exponential component
```

```
z20=eval(TF20);
z30=eval(TF30);
z40=eval(TF40);
z50=eval(TF50);
```

```
% Static plots
```

```
figure();
surfc(x,y,log(abs(z20)));
title('Plasma Laplace complex roots (20MilliTorr)');
xlabel('Real(x) (Exponentials)');
ylabel('Imag(x) (Sinusoids)');
zlabel('Log value of Transfer function');
```

```
figure();
surfc(x,y,log(abs(z30)));
title('Plasma Laplace complex roots (30MilliTorr)');
xlabel('Real(x) (Exponentials)');
ylabel('Imag(x) (Sinusoids)');
zlabel('Log value of Transfer function');
```

```
figure();
surfc(x,y,log(abs(z40)));
title('Plasma Laplace complex roots (40MilliTorr)');
xlabel('Real(x) (Exponentials)');
ylabel('Imag(x) (Sinusoids)');
zlabel('Log value of Transfer function');
```

```
figure();
surfc(x,y,log(abs(z50)));
title('Plasma Laplace complex roots (50MilliTorr)');
xlabel('Real(x) (Exponentials)');
ylabel('Imag(x) (Sinusoids)');
zlabel('Log value of Transfer function');
```

```
% Now for a little animation
```

```
figure();
h=surfc(x,y,log(abs(z20)));
xlabel('Real(x) (Exponentials)');
ylabel('Imag(x) (Sinusoids)');
zlabel('Log value of Transfer function');
%for i=1:100 % for video capture
for i=1:1
```

```
set(h,'ZData',log(abs(z20)));  
title('Plasma Laplace complex roots (20MilliTorr)');  
drawnow  
pause(.5);
```

```
set(h,'ZData',log(abs(z30)));  
title('Plasma Laplace complex roots (30MilliTorr)');  
drawnow  
pause(.5);
```

```
set(h,'ZData',log(abs(z40)));  
title('Plasma Laplace complex roots (40MilliTorr)');  
drawnow  
pause(.5);
```

```
set(h,'ZData',log(abs(z50)));  
title('Plasma Laplace complex roots (50MilliTorr)');  
drawnow  
pause(.5);
```

```
end
```

Appendix T Laplace Transform Video Pressure Transfer Functions

The following video shows the variation in the solutions for the Oxford Instruments PlasmaLab 100 CCP transfer function as the pressure is swept from 20 to 30 to 40 to 50 millitorr.



Complexroot_TF_20_30_40_50millitorr.mp4 (Command Line)

Appendix U Netlist for Geometric Resonance and Hot & Cold Electron Injection

The following is a netlist for Spice to generate the Geometric Resonance and Hot and Cold Electron Injection for the Oxford Instruments PlasmaLab 100 CCP

```
* C:\LTSpice\Varactor_TunedLC_Resonances_Geo_Resonance\circuit1.asc
C2 applied N007 50pF
R1 N007 N014 3R
C3 N016 0 200pF
R2 N019 N001 .5R
V1 N019 0 SINE(0 150V 13.5596e6 0 0 0) AC 150 20 Rser=50 Cpar=0
R7 N015 Vant 1Meg tol=1 pwr=0.1
R39 applied N002 .1R
R40 N015 N016 3R
I1 N018 0 PULSE(0 70m 46n 1n 1n 24n 73.74848816n)
I2 N018 0 PULSE(0 60m 32n 1n 1n 8n 73.74848816n)
C36 N002 N001 176pF
I3 N018 0 PULSE(0 40m 22n 1n 1n 5n 73.74848816n)
I4 N018 0 PULSE(0 100m 5n 50p 50p 9n 73.74848816n)
I5 applied N003 PULSE(0 100m 5n 50p 50p 9n 73.74848816n)
I6 applied N003 PULSE(0 40m 22n 1n 1n 5n 73.74848816n)
I7 applied N003 PULSE(0 60m 32n 1n 1n 8n 73.74848816n)
I8 applied N003 PULSE(0 70m 46n 1n 1n 24n 73.74848816n)
I9 N004 applied PULSE(0 60m 0n 50p 50p 8n 73.74848816n)
I10 N004 applied PULSE(0 80m 15n 50p 50p 21n 73.74848816n)
I11 N004 applied PULSE(0 100m 44n 50p 50p 10n 73.74848816n)
I12 N004 applied PULSE(0 50m 64n 50p 50p 5n 73.74848816n)
R4 N003 N007 .1R
R5 N004 N007 .1R
R41 N016 N018 .1R
R43 N016 N017 .1R
I13 0 N017 PULSE(0 60m 0n 50p 50p 8n 73.74848816n)
I14 0 N017 PULSE(0 80m 15n 50p 50p 21n 73.74848816n)
I15 0 N017 PULSE(0 100m 44n 50p 50p 10n 73.74848816n)
I16 0 N017 PULSE(0 50m 64n 50p 50p 5n 73.74848816n)
L1 N014 N015 {PlasmaL1}
I18 N021 0 PULSE(0 60m 37n 500p 500p 8n 73.74848816n)
I19 N021 0 PULSE(0 100m 18n 2n 8n 10n 73.74848816n)
I22 applied N005 PULSE(0 100m 18n 2n 8n 10n 73.74848816n)
I23 applied N005 PULSE(0 60m 37n 500p 500p 8n 73.74848816n)
I26 N006 applied PULSE(0 80m 2n 500p 2n 15n 73.74848816n)
I27 N006 applied PULSE(0 110m 52n 50p 50p 21n 73.74848816n)
R3 N005 N007 .1R
R8 N006 N007 .1R
R9 N016 N021 .1R
R10 N016 N020 .1R
I30 0 N020 PULSE(0 8m 2n 500p 2n 15n 73.74848816n)
I31 0 N020 PULSE(0 11m 52n 50p 50p 21n 73.74848816n)
D1 applied N007 D
D2 0 N016 D
R6 N015 0 100R
R11 N009 N008 100R
L2 N010 N009 4.51175e-07
C1 N011 N010 7.67292e-13
```

```

C4 N011 0 1.08933e-10
L3 N011 0 3.17795e-09
L4 N012 N011 1.08933e-06
C5 N013 N012 3.17795e-13
C6 N013 0 4.51175e-11
L6 N013 0 7.67292e-09
R12 N013 0 100R
R13 N008 N014 {RSWITCH}
.model D D
.lib C:\Users\Dave\Documents\LTspiceXVII\lib\cmp\standard.dio
.tran 0 24000ns 12u 100ps
* Comment .step param PlasmaL2 list 35nH 2uH 30uH
* COLD Electrons
* HOT Electrons
* Upwards

```