

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

<http://wrap.warwick.ac.uk/130616>

Copyright and reuse:

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it.

Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

3904

**Quantitative Digital Image Processing in
Fringe Analysis and Particle Image
Velocimetry (PIV)**

Volume I

Thomas Richard Judge
Department of Engineering,
University of Warwick

This thesis is submitted for the degree of
Doctor of Philosophy

July 22, 1992

Declaration

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been done by myself except where stated otherwise.

Thomas Richard Judge

Acknowledgements

I would like to give special thanks to Dr. Peter Bryanston-Cross, my supervisor, for his constant support and advice during the past three years. I would like to acknowledge the help and support of the other members of our research team; Dr. David Towers, Cathy Towers and Quan Chenggen.

I would like to thank Dr. Alan Gibbons and Darren Kerbyson for advice concerning parallel approaches to the minimum spanning tree problem.

I would also like to thank those who aided in proof reading this thesis; Louise Auchterlonie, Kay Garbett, Ruth Judge, Marcelo Funes-Gallanzi, Tim Pearce and Darren Kerbyson.

Thomas Richard Judge

To Mum, Dad, Ruth and Louise

Abstract

This thesis concerns the application of Quantitative Digital Image Processing to some problems in the domain of Optical Engineering. The applications addressed are those of automatic two dimensional phase unwrapping and the analysis of images from high speed particle image displacement velocimetry.

The first application involves subdivision of the two dimensional image of a wrapped phase map into small two dimensional areas or tiles, which are unwrapped individually, in order that discontinuities may be localised to small areas. In this case the discontinuities have a contained effect on the unwrapped phase solution.

The concept of minimum spanning trees, from Graph Theory, is employed to minimise the effect of such local discontinuities by computation of an unwrapping path which avoids areas likely to be discontinuous in a probabilistic manner. This approach is implemented over two hierarchical levels, the first level identifying pixel level discontinuities such as spike noise, the second addressing larger scale discontinuities which may not be detected by pixel level comparisons, but which can be detected by comparison of the local solutions of image areas larger than the pixel.

The second application is in the area of Particle Image Displacement Velocimetry (PIDV). A digital processing method is developed for high speed PIDV. In high speed PIDV the seeding is sparsely distributed. This method attempts to pair individual particle images, rather than statistically average the positions of a large number of particle images as is the case with other analysis methods. The digital processing method is suitable for use with Video PIDV whose feasibility has recently been demonstrated.

Contents

1	Introduction: Optical Engineering and Image Processing	19
1.1	Optical Engineering	19
1.2	Digital Image Processing	21
2	Fringe Analysis	25
2.1	Introduction	25
2.1.1	Interference	26
2.1.1.1	A Note on Notation	27
2.1.1.2	Light as a Transverse Wave	27
2.1.1.3	The Interference of Light Waves as Scalars	29
2.1.1.4	Conditions for Interference	30
2.1.2	Producing Fringe Patterns	31
2.1.3	Example From Real Time Holographic Interferometry	35
2.1.4	Applications	35
2.1.5	History Of Fringe Analysis	37
2.2	The Fringe Analysis Problem	38
2.3	The Analysis Of A Conventional Interferogram By Fringe Tracking	43
2.3.1	Problems Associated With Fringe Tracking In Completing The Interferogram Analysis	44
2.4	Electronic And Quasi Heterodyning	46
2.4.1	Electronic Heterodyning	46
2.4.2	Quasi-Heterodyning	49
2.5	The Fourier Transform Technique	53
2.6	Phase Unwrapping Algorithms	62

2.6.1	The Phase Fringe Counting/Scanning Approach To Phase Unwrapping	62
2.6.2	Cellular-automata Method for Phase Unwrapping . . .	63
2.6.2.1	Detecting Possible Inconsistencies	64
2.6.2.2	Two-Dimensional Algorithm Description . . .	65
2.6.2.3	Cellular-automata Conclusion	66
2.6.2.4	Modification to Original Cellular Automata Algorithm	68
2.6.3	Berlin Development of Minimum Spanning Tree Method	69
2.6.3.1	The Berlin Pixel Level Minimum Spanning Tree Algorithm	70
2.6.3.2	Conclusion to the Berlin Pixel Level Minimum Spanning Tree Algorithm	70
2.6.4	Noise-immune Cut Methods of Phase Unwrapping . . .	71
2.6.4.1	Cut Method Conclusion	76
2.6.5	Phase Unwrapping by Regions	76
2.6.5.1	Region Method Conclusion	78
2.6.6	Phase Unwrapping Using a Priori Knowledge about the Band Limits of a Function	78
2.6.6.1	The Band Limited Phase Unwrapping Algorithm	81
2.6.6.2	Band Limited Phase Unwrapping Algorithm Conclusion	81
2.7	Conclusion	81
3	The Minimum Spanning Tree Approach to Phase Unwrapping	89
3.1	Introduction	89
3.1.1	The Processing Objective	90
3.1.2	The Processing Problem	91
3.2	Tests for Low Modulation and Absence of Carrier	92
3.3	Examples of Discontinuity Types	94
3.3.1	Aliasing Induced	94
3.3.2	Hole in Object	94

3.3.3	Absence of Carrier	94
3.4	Adoption Of The MST Tiling Method	94
3.5	Graphs in Graph Theory	101
3.5.1	The Concept of Connectedness	101
3.5.2	Trees	103
3.5.3	Minimum Spanning Trees	103
3.6	Hierarchical Phase Unwrapping Using Minimum Spanning Trees	104
3.7	High Level Phase Unwrapping, Connection of Unwrapped Tiles	104
3.7.1	Factors for Assessing the Quality of Data in the Field at the Tile Level	105
3.7.1.1	Agreement of Neighbouring Tiles	105
3.7.1.2	Points of Low Modulation	106
3.7.1.3	Fringe Density	106
3.7.1.4	Fringe Edge Termination Points	106
3.7.1.5	Metrics	109
3.7.2	The Minimum Spanning Tree Algorithm	110
3.7.3	Selection of an Appropriate Tile Size	112
3.8	Low Level Phase Unwrapping, Pixel to Pixel	116
3.8.1	The Calculation of Edge Weights at the Pixel Level . .	117
3.8.2	The Interaction of Tile Level Phase Unwrapping with the Pixel Level	119
3.9	Examples of Tile Level Phase Unwrapping	122
3.9.1	FFT Example of Tile Level Phase Unwrapping	122
3.9.2	Phase Stepping Example of Tile Level Phase Unwrapping	132
3.10	Conclusion	138
4	Image Capture and Processing	149
4.1	The CCD Camera	149
4.2	Capturing the Image of a Scene	152
4.2.1	The Effect of Intensity Quantization	153
4.3	An Experiment to Investigate the Distribution of Noise in a Particular Camera/Digitiser Pair	158
4.3.1	Effect of the Camera Response Upon the Accuracy of Fringe Analysis Techniques	165

4.3.2	Image Data Format	166
4.4	Spatial Smoothing	166
4.4.1	Discrete Linear Operator	167
4.4.2	Nonlinear Operator	168
4.4.2.1	Median Filters	169
4.4.2.2	Median Filter Properties	170
4.5	Example of Smoothing on a Speckle Image	171
4.5.1	Speckle	171
4.5.2	Electronic Speckle Patterns	172
4.6	Edge Detection	175
4.7	Industrial Requirements for a Fringe Analysis System	183
4.8	System Development by Other Researchers in Fringe Analysis	187
4.9	The Development Process for the FRAN Fringe Analysis System	189
4.10	High Speed Sequential Microprocessors	190
4.11	Parallel Processing	191
4.12	Fringe Analysis in Parallel	193
4.13	Parallel Minimum Spanning Tree Algorithm for Phase Un- wrapping and its Implementation	196
4.14	Conclusion	200
5	Example Fringe Analysis Applications	205
5.1	Introduction	205
5.2	Deformation Measurement of a Metal Disc	205
5.3	Introduction to Disc Deformation	206
5.4	Theory	207
5.4.1	Direct Deformation Measurement of a Disc	207
5.4.2	Objective of Holographic Carrier Fringe Technique	207
5.5	Image Processing Technique	208
5.5.1	Windowing to Isolate a Side Lobe	208
5.6	Experimental Description and Results	209
5.6.1	Experimental Set-up	209
5.6.2	Correction for the Non-linearity of the Carrier Fringes	211
5.6.3	Results and Evaluation	213
5.7	Conclusion of Disc Deformation Measurement	225

5.8	Holographic Flow Visualisation	226
5.9	Analysis of the Modes of Vibration of a Vibrating Board by Phase Stepping	231
5.10	Soldering Iron, Aliasing on An Object In The Field	239
5.11	Conclusion	253
6	Particle Image Displacement Velocimetry	256
6.1	Introduction	256
6.2	History of Particle Image Displacement Velocimetry	263
6.3	Processing Methods	264
6.3.1	Two Dimensional Correlation and Two-Dimensional Spectrum Analysis of Young's Fringe Pattern	265
6.4	Initial Semi-Automatic Data Reduction System for Transonic PIDV via Spatial Pairing	270
6.5	Improved Spatial Pairing PIDV Analysis System	278
6.5.1	The Problem of Ambiguous Pairing and Comparison of Particle Positions	280
6.6	Example of PIDV Analysis by 2D Autocorrelation Technique on a Single Pair of Particles	283
6.7	The 2D Autocorrelation Method Applied With and Without Preprocessing of the PIDV Image	287
6.8	Comparison of Analysis by 2D Autocorrelation with that by Spatial Pairing	299
6.9	Discussion of Results for Spatial Pairing and Correlation Tech- niques	305
6.10	An Example of High Speed PIDV at a Large Stand Off Dis- tance (Applied in the Transonic Wind Tunnel of ARA Bedford)	307
6.11	Conclusion	315
7	Conclusion	319
A	Documentation For FRANSYS Fringe Analysis System	324
A.1	Introduction	324
A.2	Sunview Window Interface	325
A.2.1	TIF Image File Display	325

A.2.2	Pseudo Colour	328
A.2.3	Grey Scale	328
A.2.4	Printing an Image File	328
A.2.5	Reading a Project File	329
A.2.6	Creating a Project File	329
A.2.7	Converting To Sun Raster File Format	330
A.2.8	Removing an Image From the Canvas	330
A.3	The FRAN Program	330
A.3.1	Project File Format	331
A.3.2	An Example Project File	344
A.4	Converting To TIF Format	344
A.4.1	Defaults for Mktif	345
A.5	Converting From TIF Format	346
A.6	Creation of FFT Test Image	346
A.6.1	Defaults for Mkfft	347
A.7	Displaying Information about a TIF Image	349
A.7.1	Example Use of Tifinf	349
 B Documentation for the Automated PIDV Image Analysis		
Package (AP)		352
B.1	Introduction	352
B.2	User Interface	353
B.3	Main Menu	353
B.3.1	File Tif	353
B.3.2	File Piv	354
B.3.3	View Tif	354
B.3.4	View Vec	355
B.3.5	Zoom In	355
B.3.6	Filter	355
B.3.6.1	Image Scaling	356
B.3.6.2	Threshold Intensity Level for 8 Bit TIF File .	356
B.3.6.3	Particle Size	357
B.3.6.4	Feature Size	357
B.3.6.5	Pulse Separation	357

B.3.6.6	Minimum / Maximum Velocity Accepted . . .	357
B.3.6.7	Minimum / Maximum Angle Accepted . . .	359
B.3.7	Process	359
B.3.8	Batch	362

List of Figures

2.1	A Transverse Wave Travelling in the Z Direction	27
2.2	Disturbance Resolved into Components along Two Mutually Perpendicular Axes	28
2.3	Twyman and Green's Interferometer	32
2.4	The Interference of a Plane and Perturbed Wavefront. (a) The Two Wave Fronts. (b) The Interference Fringe Pattern Generated. (The intensity profile of the fringes generated is sinusoidal)	33
2.5	Example Wrapped Map	40
2.6	Relationship of Elements in Sector of Fringe Analysis Considered	42
2.7	Example Problem in the Determination of Fringe Orders in a Discontinuous Field	45
2.8	Heterodyned Twyman and Green Interferometer	47
2.9	Interferometer Used in Quasi-Heterodyning	50
2.10	Computer Generated Example of Interferogram (The Subject Simulated is a Circular Disc with Pressure Applied at its Centre to Induce Deformation)	54
2.11	Computer Generated Example of Interferogram (The Subject Simulated is a Tilted Flat Plate)	56
2.12	Computer Generated Example of Interferogram (The Subject Simulated is a Centrally Deformed Plate which has been Tilted between Exposures)	57
2.13	Real Holographic Interferogram of Centrally Deformed Plate	58
2.14	Separated Spectra of Fringe Pattern	59
2.15	One of the Side Lobes Translated to the Origin	60

2.16	2 by 2 Pixel, Path Consistency Check	65
2.17	A Positive Residual	72
2.18	Two Alternative Pixel Paths for Unwrapping the Phase at Data Point x_1, y_1	73
2.19	Example Cut Made Between Two Discontinuity Sources s $= +1$ and $s = -1$	75
2.20	Field Divided into Regions	77
3.1	Example of Aliasing Induced Discontinuity, (Original Inter- ferogram)	95
3.2	Example of Aliasing Induced Discontinuity, (Wrapped Phase Map by FFT Method)	96
3.3	Example of Discontinuity Introduced by Hole in Object, (Electronic Speckle Interferogram after Prefiltering)	97
3.4	Example of Discontinuity Introduced by Hole in Object, (Wrapped Phase Map by Phase Stepping with 3 Images)	98
3.5	Example of Discontinuity Introduced by Absence of Carrier, (Original Interferogram)	99
3.6	Example of Discontinuity Introduced by Absence of Carrier, (Wrapped Phase Map by FFT Method)	100
3.7	Example of a Graph	102
3.8	A Disconnected Graph	102
3.9	An Example of a Tree	103
3.10	A Minimum Spanning Tree of the Weighted Graph	104
3.11	Tiled Section of the Fringe Field Corresponding to the Weighted Graph	105
3.12	Neighbouring Tiles Showing Overlap	105
3.13	Edge Termination Points	107
3.14	Tiles being Arranged at their Correct Height Offsets	108
3.15	Selection of Tile Size: Tile with One Broken Fringe Edge	112
3.16	Selection of Tile Size: Tile with Two Broken Fringe Edges	114
3.17	Illustration of Effect of Tile Size	115
3.18	Pixel Level, Computation of Edge Weights. Weights are Computed over Four Pixels for Each Pair	118

3.19	Mountaineer Analogy in Pixel Level Phase Unwrapping . . .	119
3.20	Interaction of Tile Level Phase Unwrapping with the Pixel Level	120
3.21	Pixel Level Phase Unwrap Circumnavigates Phase Distortion	121
3.22	Tile Level Phase Unwrap Circumnavigates Smooth Area . .	121
3.23	Wrapped Phase Map for Disc Showing Problem Areas . . .	123
3.24	Points of Low Modulation for Disc (shown in white) . . .	124
3.25	Weighting Factor for Disc : Agreement between Neighbour- ing Tiles	125
3.26	Weighting Factor for Disc : Low Modulation Points	126
3.27	Weighting Factor for Disc : Fringe Edge Termination Points	126
3.28	Weighting Factor for Disc : Points on Fringe Edges	127
3.29	Combined Weighting Factors for Disc	128
3.30	Edge Detection and Minimum Spanning Tile Tree for Disc .	129
3.31	Contour Plot of Unwrapped Numerical Phase Data (Sam- pled at Every 5th Pixel in X) Showing Circumvention of Shadow Discontinuity	130
3.32	Raw ESPI Image	133
3.33	Wrapped Phase Map for Chamber Showing Problem Areas .	134
3.34	Points of Low Modulation for Chamber (shown in white)	135
3.35	Weighting Factor for Chamber : Agreement between Neigh- bouring Tiles	138
3.36	Weighting Factor for Chamber : Low Modulation Points . .	139
3.37	Weighting Factor for Chamber : Fringe Edge Termination Points	139
3.38	Weighting Factor for Chamber : Points on Fringe Edges . .	140
3.39	Combined Weighting Factors for Chamber	141
3.40	Edge Detection and Minimum Spanning Tile Tree for Cham- ber (Includes Low Modulation Points in Grey)	142
3.41	Contour Plot of Unwrapped Numerical Phase Data for Cham- ber (Sampled at Every 5th Pixel) Showing Circumvention of Missed Fringe Edges	143

3.42	Mesh Plot of Unwrapped Numerical Phase Data for Chamber after Post Processing via 3 by 3 Median Filter (Sampled at Every 5th Pixel)	144
3.43	Grey Scale Image of Unwrapped Numerical Phase Data for Chamber Showing Circumvention of Missed Fringe Edges . .	145
3.44	Octagonal tiles would allow better Consistency Testing (across diagonals)	146
4.1	CCD Cells Arranged on Rectangular Grid	149
4.2	1D Slice for p-type CCD Cell	150
4.3	Generation of Minority Carrier Charge Packet from Incident Illumination (1-D Slice)	151
4.4	Input Video Signal Gain and Offset Control	155
4.5	Representation of Sampling the Analog Intensity Signal . .	155
4.6	Probability Density Function of Quantization Error with Truncation	157
4.7	Experimental Arrangement for Camera Noise Experiment .	158
4.8	Result of Camera/Digitiser Noise Experiment, Gain 1	162
4.9	Result of Camera/Digitiser Noise Experiment, Gain 2	163
4.10	Result of Camera/Digitiser Noise Experiment, Gain 3	164
4.11	Standard Deviation of Intensity Against Mean for Varying Gain	164
4.12	Section of an Electronic Speckle Pattern (Normalised Intensity)	172
4.13	Section of an Electronic Speckle Pattern (After 3 X 3 Average)	173
4.14	Section of an Electronic Speckle Pattern (After 3 X 3 Median)	174
4.15	Illustration of Step Edge Detection using Zero Crossing of 2nd Derivative, ($f(x)$ is a step function, $f_2(x)$ is a smoothed step function)	178
4.16	Typical Curves for Different Edge Operators	179
4.17	Gaussian Smoothing Kernel (Sigma = 5)	180
4.18	Laplacian-of-a-Gaussian Smoothing Kernel (Sigma = 5) . .	181

4.19	Example Wrapped Phase Map for Edge Detection	184
4.20	Sobel Gradient Edge Detector with Adaptive Thresholding Applied to Wrapped Phase Map, Kernel = 3 X 3	185
4.21	Laplacian of a Gaussian Edge Detection Applied to Wrapped Phase Map, Kernel = 9 X 9, Sigma = 1.0	186
4.22	Mapping of Problem to Algorithm, Algorithm to Architecture	192
5.1	Experimental Arrangement for Recording Holograms of the Disc by the Carrier Fringe Technique	210
5.2	Optical and Electronic Arrangement for Reconstruction of the Holograms	211
5.3	Schematic Diagram for Generation of Carrier Fringes at Slight Inclination	212
5.4	Double-exposure Holographic Interferogram of the Centrally Loaded Disc	214
5.5	Holographic Carrier Fringe Interferogram of the Centrally Loaded Disc	215
5.6	Wrapped Phase Map for Centrally Loaded Disc, Generated by the FFT Technique	216
5.7	Normalised grey scale plot of deformation produced by un- wrapping procedure, before correction for non linearity of carrier fringes.	217
5.8	(a) Digitised intensity data of central raster in the interfer- ogram; (b) intensity data weighted by Papoulis window. . .	218
5.9	(a) Power spectrum of central raster from the interferogram with carrier and deformation (the window is indicated by the dashed dot line); (b) side lobe translated by the carrier frequency to the origin position.	219
5.10	3-D perspective plot of the out of plane displacement of the centrally loaded disc after phase unwrapping and correction.	220
5.11	Side view of unwrapped numerical phase data.	222
5.12	Side view of the unwrapped and corrected numerical phase data.	223

5.13	Comparison between the theoretical and measured deformation for cross section.	224
5.14	Finite Fringe Hologram of a NACA 0012 Aerofoil at a Freestream Mach Number of 0.8, copyright British Aerospace PLC 1991 for publication	227
5.15	Wrapped Phase Map by 1D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	228
5.16	Edge Detection and Tile Connection Tree via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	229
5.17	Grey Scale Unwrapped Phase Map by 1D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	230
5.18	Averaged Background Image for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	232
5.19	Wrapped Phase Map by 2D FFT Method for NACA 0012 Aerofoil at a Freestream Mach, copyright British Aerospace PLC 1991 for publication	233
5.20	Edge Detection and Tile Connection Tree via FRAN for 2D FFT Analysis of NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	234
5.21	Contour Plot of Unwrapped Phase Map, Every 5th Pixel, by 2D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication . . .	235
5.22	Mesh Plot of Unwrapped Phase Map, Every 5th Pixel, by 2D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication	236
5.23	Grey Scale Unwrapped Phase Map by 2D FFT Method via FRAN for Finite Fringe Hologram of a NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication . . .	237
5.24	Original Holographic Cosinusoidal Interferogram of Vibrating Board after Prefiltering	240
5.25	Wrapped Fringe Field for Vibrating Board	241

5.26	Low Modulation Noise (in White) for Vibrating Board . .	242
5.27	Edge Detection of Wrapped Fringe Field for Vibrating Board, Showing Tiles and Connection Tree	243
5.28	Grey Scale Unwrapped Fringe Field for Vibrating Board . .	244
5.29	Contour Plot of Unwrapped Phase for Vibrating Board, Ev- ery 5th Pixel on Long Axis	245
5.30	Mesh Plot of Unwrapped Phase for Vibrating Board	246
5.31	Soldering Iron Interferogram with Carrier Fringes	247
5.32	Wrapped Phase Map Of Soldering Iron	248
5.33	Edge Detection and Tile Connection Tree for Soldering Iron	249
5.34	Grey Scale Plot of Solution Showing Circumvention of Dis- continuities in Area of Soldering Iron	250
5.35	Contour Map Of Soldering Iron Solution	251
5.36	Mesh Plot Of Soldering Iron Solution	252
6.1	Coding Techniques for Image Velocimetry	259
6.2	Schematic Diagram of PIDV as Applied in Transonic Wind Tunnel	259
6.3	Interrogation of a Double Exposed Transparency (a) Pro- cessing in the 2-D Image Plane via Two Dimensional Cor- relation. (b) Processing by Young's Fringe Analysis in the Fourier Transform Plane of the Images	266
6.4	Efficient Use of Spatial Correlation	269
6.5	Initial Semi-Automatic Data Reduction System for Tran- sonic PIDV	271
6.6	Image Captured from Original Data Reduction System show- ing Two Clear Particle Pairs	272
6.7	Computation of Particle Centre from Bounding Box in PIDV	273
6.8	Error in Measuring Velocity Vector at a Variety of Angles .	275
6.9	Error in Vector Measurement is Greatest at 45 Degrees . . .	276
6.10	Data Reduction System Based Upon Flat Bed Scanner . . .	278
6.11	Schematic of Particle Pairs Showing an Ambiguous Pairing Problem	280

6.12	Schematic of Particle Pairs, Circles Define the Maximum Distance that a Particle could have Moved, Pulse to Pulse .	280
6.13	Schematic of Particle Pairs, Ambiguous Pairings are Deleted and Velocities Computed for the Remaining Pairs	281
6.14	Bounding Box Test to Improve Performance of Pairing Algorithm	283
6.15	Single Pixel Pair Image	284
6.16	Single Pixel Pair Image 2D Power Spectrum with DC Removed	285
6.17	Single Pixel Pair Autocorrelation	286
6.18	Thrust Reverser	288
6.19	Whole Field View of PIDV Experiment on Thrust Reverser showing Area of Test Image in Box	289
6.20	Test Image	290
6.21	Test Image 2D Power Spectrum with DC Removed	291
6.22	Test Image 2D Autocorrelation (Circle Defines Maximum Velocity 300 m/s)	292
6.23	Test Image after Reducing Particle Images to Single Pixels .	294
6.24	Reduced Test Image 2D Power Spectrum	295
6.25	Reduced Test Image 2D Autocorrelation (Circles Define Minimum Velocity = 100 m/s and Maximum Velocity = 300 m/s)	296
6.26	Contour Plot of Central Region of Autocorrelation for Reduced Test Image	297
6.27	Mesh Plot of Central Region of Autocorrelation for Reduced Test Image	298
6.28	Part of Test Image after Reducing Particle Images to Single Pixels	300
6.29	Part of Reduced Test Image 2D Power Spectrum with DC Removed	301
6.30	Part of Reduced Test Image 2D Autocorrelation (Circles Define Minimum Velocity = 100 m/s and Maximum Velocity = 300 m/s)	302
6.31	Contour Plot of Central Region of Autocorrelation for Part of Reduced Test Image	303

6.32	Mesh Plot of Central Region of Autocorrelation for Part of Reduced Test Image	304
6.33	Result of Test Image Analysis by Spatial Analysis Superim- posed on Test Image	306
6.34	PIDV Vector Plot For Thrust Reverser, Pulse Separation 2.0 Microseconds, Minimum Allowed Velocity = 100 m/s, Maximum Allowed Velocity = 300 m/s, Minimum Allowed Angle (from vertical) = 120 degrees, Maximum Allowed Angle = 170 degrees	308
6.35	PIDV Histogram of Velocity for Thrust Reverser	309
6.36	PIDV Histogram of Velocity Vector Angle for Thrust Reverser	310
6.37	Scan of PIDV Print, Mach No. 0.2, Frame No. 7, 10 Mi- crosecond Pulse Separation, Model at Incidence of 5 Degrees	311
6.38	PIDV Vector Plot Mach No. 0.2, Frame No. 7, 10 Microsec- ond Pulse Separation, Model at Incidence of 5 Degrees . . .	312
6.39	PIDV Histogram of Velocity, Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees	313
6.40	PIDV Histogram of Velocity Vector Angle, Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees	314
A.1	Example FFT Image Generated By mkfft	326
A.2	Sunview Window Interface to FRANSYS	327
A.3	Image Data Display in Sunview	328
A.4	Example FFT Wrapped Map	334
A.5	Example FFT Edge Detection of Wrapped Map, With Tree Added	335
A.6	Example FFT Unwrapped Map	336
A.7	Original Intensity Data For Raster 128	338
A.8	Intensity Data of Raster 128 Weighted by Papoulis Window	339
A.9	Power Spectrum of Scan Line and Papoulis Weighting Window	339
A.10	Translated Side Lobe	340
A.11	Example FFT Mesh Plot	341

A.12	Example FFT Contour Plot	342
A.13	Boundary Example	343
A.14	Example Project File	344
A.15	Example Concentric Fringe Image	348
A.16	Tifinf Example	350
B.1	PC Screen Display for AP Package	353
B.2	Schematic of Particle Pairs Showing an Ambiguous Pairing Problem	358
B.3	Schematic of Particle Pairs, Circles Define the Maximum Distance that a Particle could have Moved, Pulse to Pulse .	358
B.4	Schematic of Particle Pairs, Ambiguous Pairings are Deleted and Velocities Computed for the Remaining Pairs	358
B.5	Specifying Band Limits for Direction	359

List of Tables

3.1	Tabular Representation of the Weighted Graph	111
4.1	Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 1 . .	160
4.2	Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 2 . .	161
4.3	Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 3 . .	161
4.4	Summary of Goodness of Fit Tests, Comparing the Noise Distribution with the Normal Distribution	162
4.5	Execution Times for Breuckmann's Dedicated Software+Hardware Combination	189
4.6	Execution Times for Hardware Independent Analysis Soft- ware, on Various Processors, Using 3 Phase Stepped (512 by 512) Images, Without Prefiltering	189
6.1	Summary of Fourier Analysis Result for Edited Test Image .	299
6.2	Summary of Fourier Analysis Result for Lower Left Corner of Test Image	299
6.3	Summary of Spatial Pairing Analysis Result for Test Image	299
6.4	Summary of Spatial Pairing Analysis Result for Thrust Re- verser (Complete Field)	307
6.5	Summary of Spatial Pairing Analysis Result for Example ARA Test	309

Chapter 1

Introduction: Optical Engineering and Image Processing

1.1 Optical Engineering

Optical engineering is an established field; it is a generic term which covers all applications where a measurement is being made or processing performed using an optical element or aspect. It plays an ever increasing role in a wide range of applications. This expansion has been due in part to the discovery and rapid development of versatile lasers covering the ultraviolet to far-infrared regions, and a parallel development of solid-state and optical materials. The introduction of optical fibres and semiconductor lasers in communication has also stimulated the relationship between optics and communication engineering [1]. Other important areas within optical engineering are in optical measurement, and optical diagnostics. These are respectively, quantitative and qualitative techniques. The extraction of quantitative measurements from optically captured data, via image processing, is the subject of this work.

There are a number of important issues in the development of an optically based measurement technique. Firstly the nature of the parameter to be measured must be considered, then the light source, the optical components (lens etc.), the effect of the experimental environment, consideration of

the process which is to record the result, and then analysis of the recording. The subject is multidisciplinary by its nature, requiring knowledge from such areas as Physics, Mechanical Engineering and Computer Science. Physics is required to cover the theoretical aspects. Mechanical Engineering is involved in the design and development of experiments. Computer Science is involved in the design and development of the analysis techniques and also in providing digital control of experiments.

There are several motivations for the use of optical techniques. Firstly, the properties of light make highly accurate measurement possible. Secondly, optical techniques are non-contact, which is important for many applications where a measurement would be upset by touch, or the introduction of a foreign body, for example in flow visualisation. In addition optical techniques are non-destructive.

Optically captured results (stored either holographically, photographically or by video) need further processing to yield quantitative data. Computer Science covers the techniques used to develop the necessary algorithms, and the machine architectures upon which such algorithms could be efficiently implemented.

Historically there have been several impediments to the quantitative analysis of many images generated by optical techniques. In many applications quantitative analysis has had to attend the development of low-cost high-speed digital computers, the development of low-cost imaging technology, which could be easily interfaced to such computing facilities, the development of the discipline of image processing and standard methods, and lastly the development of the specific methods of analysis required for the image types found in optical engineering.

This thesis concerns the development of a number of analysis strategies, aimed at quantitative analysis of two image subclasses in optical engineering. The first subclass being interferometric images, the second subclass being high speed images from Particle Image Displacement Velocimetry (PIDV). It has been necessary to apply techniques from the field of Digital Image Processing and more general computer science to achieve the ends of analysis.

There is often an interdependency between the design of the experiment and the design of the analysis technique. In the development of the complete

system, one of the aims is to simplify this relationship. The development of analysis techniques requires an understanding of the experiment and the devices used to capture the image data, as well as the techniques employed in analysis. There has been some direct interaction in the experiments themselves and digital control has been provided in several instances.

1.2 Digital Image Processing

Digital image processing is a relatively new field. It has experienced tremendous growth over the past decade. There has been some development of standard algorithms for defined tasks. For example, edge detection [2]. The wider goal of computer vision, however, is to solve the 'vision problem' as a whole. That is in the sense of scene analysis, or robot vision, to produce a description of the scene being observed, in terms of the objects in view and their properties or at a higher level what the scene as a whole means. There has been less success in this area.

The last decade has seen an increasing effort to develop sophisticated, real-time automatic image processing systems.

The processing tasks considered in this work require the extraction and display of coded information from an image, rather than emulation of human behaviour or image enhancement. Take the case of Interferometry. In this application an interference pattern is produced from beams of light, influenced by a physical parameter, e.g. temperature. The interference pattern encodes the temperature information and an image processing function is needed to extract it. This is typical of many Engineering applications where images are encoded, perhaps by physical effects during an experimental process. Structured light is often employed to encode information.

In pursuing the objectives of image processing, a range of often interdependent numerical tools are currently employed. They include signal analysis, geometry, linear algebra, estimation theory, statistical pattern recognition, syntactic pattern recognition (for image structural description), discrete mathematics and a number of topics often referred to under the heading of "artificial intelligence", which include knowledge representation and manipulation, constraint satisfaction, symbolic manipulation, and more

recently neural networks.

The implementation requirements of image processing systems, in terms of hardware and software are substantial, particularly when "real-time" operation is needed and the resolution of images from video systems continues to increase, thus aggravating the problem. Processing capacity has placed a limit on the applications to which image processing might be applied. The possibility of real time response is an attractive prospect in many applications, particularly in Engineering, as it opens up the possibilities of human interaction with the system. This is a tremendous contrast with conventional photography, which involves time consuming wet processing.

In order to achieve high processing speeds, dedicated hardware is sometimes employed. This is virtually impossible to update with changing requirements. Software solutions offer flexibility against a reduced processing rate. The power of sequential processors continue to increase, and for many applications provide adequate solutions. Image processing and computer vision research serves as an impetus for the development of new processing architectures, particularly parallel processing systems. Such systems offer software control and therefore retain some flexibility, although standards have been slow to emerge.

A hardware independent approach to development has been taken in this work, where possible, as the test cycle of algorithmic development has required flexibility. As a by product of this strategy, the Fringe Analysis System (FRANSYS) has become a portable software application.

High speed sequential processors have developed since the inception of the project, particularly RISC (Reduced Instruction Set Computer) processors. It is seen that for many current interferometric applications, a sequential software implementation upon such a processor is adequate, and provides a flexible upgrade path. However, it is also shown that the automated phase unwrapping algorithm, developed in this work, is extremely well suited to parallel implementation.

In the application of Particle Image Displacement Velocimetry (PIDV), optical processing has been employed for some time. However, with the advent of high resolution video capture devices, digital analysis methods are more appropriate as these avoid wet processing and offer the prospect of

real time systems. Direct simulation of the optical processing methods is numerically intensive. This, together with the sparse seeding density found in high speed PIDV, has prompted a search for a more informative and less numerically intensive digital analysis technique for high speed PIDV.

Bibliography

- [1] F. T. S. Yu, I. C. Khoo, Principles of Optical Engineering, John Wiley & Sons, Inc., 1990.
- [2] R. J. Schalkoff, Digital Image Processing and Computer Vision, John Wiley & Sons, Inc., 1989.

Chapter 2

Fringe Analysis

2.1 Introduction

This work describes a new technique in the field of Automatic Interferometric Fringe Analysis.

Reid [1] suggests that the field of Fringe Analysis is now regarded by many as a subject in its own right, rather than as a specialism within image processing or interferometry. I believe this to be the case. If the field is viewed as an independent subject, then this has the benefit of encouraging the development of concepts and methods which may be applied to the patterns generated by any one of a number of optical methods. The source of the fringe pattern then has little to do with the analysis algorithm. The development described in this work is a new phase unwrapping technique, which is referred to as the Minimum Spanning Tree Tiling (MSTT) approach. This technique, in common with other methods in fringe analysis, may be applied to many types of fringe pattern, from Interferometry to Moire.

There has been an increasing interest in the automation of Fringe Analysis over the last decade. The advent of Phase Stepping [2, 3, 4, 5] and the development of FFT techniques [6, 7] has moved the emphasis in Fringe Analysis away from fringe tracking [8, 9, 10] and towards fringe counting or scanning techniques [11, 12]. One difficulty with such scanning approaches has been their susceptibility to noise.

However, several strategies have recently been proposed which exhibit noise immunity [13, 14, 16], but which have failed to address large scale

inconsistencies in interferograms which cannot be detected at the pixel level, such as aliasing.

The new method presented forms a hierarchical noise-immune technique which addresses both spike noise and large scale discontinuities.

In order to place the work in context, it is necessary to review the concepts involved in Interferometry. For the purposes of the work described here, light is modeled in the form of scalar waves.

2.1.1 Interference

The theory of light interference, using the wave theory of light, is based on the Principle of Superposition due to Thomas Young. The following discussion is derived from Hecht [17].

The essential aspect of a propagating wave is that it is a self-sustaining disturbance of the medium through which it travels. Imagine some such disturbance Ψ moving in a positive direction x . This disturbance, for the electromagnetic light wave, represents the magnitude of the electric or magnetic field. Since the disturbance is moving, it must be a function of both position x and time t and can therefore be written as;

$$\Psi = f(x, t) \quad (2.1)$$

The Principle of Superposition states that if a wave train has a displacement Ψ_1 in a given direction at a specified point and time, and a second wave train independent of the first has a displacement Ψ_2 at the same point and time, then the instantaneous resultant displacement Ψ , due to the two waves is the algebraic sum of the separate displacements.

$$\Psi = \Psi_1 + \Psi_2 \quad (2.2)$$

The Principle of Superposition implies the absolute independence of the individual components of the resultant displacement. The term Interference is, therefore, slightly misleading as the waves do not modify each other [18].

2.1.1.1 A Note on Notation

Waves for which the profile is a sine or cosine curve are known as harmonic waves. A concise representation of harmonic waves is achieved using complex notation. For example, if φ is the phase, and A is the amplitude;

$$\Psi(x, t) = A \cos(\varphi) \quad (2.3)$$

may be expressed as

$$\Psi(x, t) = \Re[Ae^{i\varphi}] \quad (2.4)$$

or more simply, assuming that the real part is taken, as

$$\Psi(x, t) = Ae^{i\varphi} \quad (2.5)$$

2.1.1.2 Light as a Transverse Wave

Light actually behaves like a transverse wave. That is, it has an associated plane of vibration which is important to consider in certain applications.

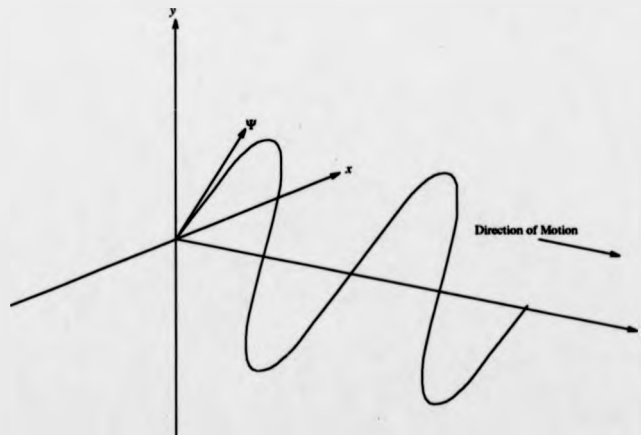


Figure 2.1: A Transverse Wave Travelling in the Z Direction

A transverse wave arises when the disturbance is perpendicular to the propagation direction. Figure 2.1 depicts a transverse wave travelling in a

direction z . In this instance, the wave motion is confined to a spatially fixed plane called the plane of vibration, and the wave is said to be linearly or plane polarised. To determine the wave completely, the orientation of the plane of vibration must be specified, as well as the direction of propagation. This is equivalent to resolving the disturbance into components along two mutually perpendicular axes, both normal to z , see Figure 2.2.

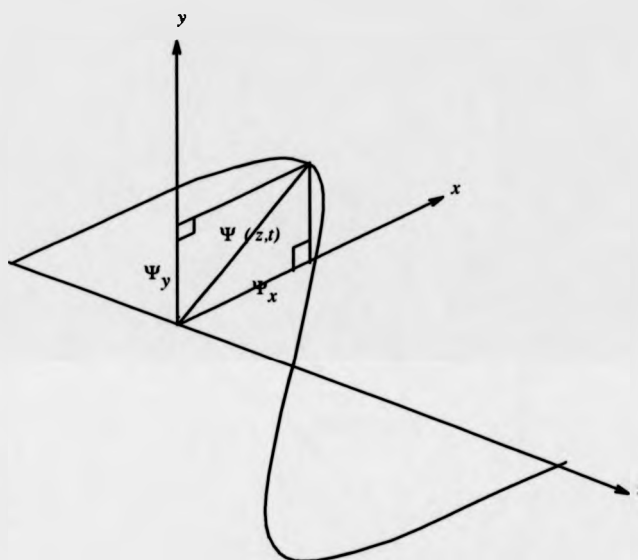


Figure 2.2: Disturbance Resolved into Components along Two Mutually Perpendicular Axes

The angle at which the plane of vibration is inclined is a constant, so that at any time Ψ_x and Ψ_y differ from Ψ by a multiplicative constant. The wave function of a transverse wave behaves somewhat like a vector quantity. With the wave moving along the z -axis;

$$\Psi(z, t) = \Psi_x(z, t)\mathbf{i} + \Psi_y(z, t)\mathbf{j} \quad (2.6)$$

where, \mathbf{i}, \mathbf{j} are the unit base vectors in Cartesian coordinates.

A scalar harmonic plane wave, moving in the z direction, is given by the expression

$$\Psi(z, t) = Ae^{i(kz \pm \omega t)} \quad (2.7)$$

where ω is the angular frequency.

An appreciation of light's vectorial nature is important. Phenomena such as optical polarisation can readily be treated in terms of this sort of vector wave picture [17]. However, there are many instances in which it is not necessary to be concerned with the vector nature of light. In particular, if the lightwaves all propagate along the same line and share a common constant plane of vibration, they may each be described in terms of one electric-field component. This approach leads to a simple and very useful scalar theory, which will be applied in this work.

2.1.1.3 The Interference of Light Waves as Scalars

Hecht [17] gives a derivation of the superposition of a pair of harmonic scalar waves, and then extends the treatment to N such waves. Given two scalar light waves, overlapping in space;

$$E_1 = E_{01} \sin(\omega t + \alpha_1) \quad (2.8)$$

and

$$E_2 = E_{02} \sin(\omega t + \alpha_2) \quad (2.9)$$

The resultant disturbance is the linear superposition of the waves. Therefore

$$E = E_1 + E_2 \quad (2.10)$$

or, expanding Equations 2.8 and 2.9;

$$E = E_{01}(\sin \omega t \cos \alpha_1 + \cos \omega t \sin \alpha_1) + E_{02}(\sin \omega t \cos \alpha_2 + \cos \omega t \sin \alpha_2) \quad (2.11)$$

If the time-dependent terms are separated out, this becomes;

$$E = (E_{01} \cos \alpha_1 + E_{02} \cos \alpha_2) \sin \omega t + (E_{01} \sin \alpha_1 + E_{02} \sin \alpha_2) \cos \omega t \quad (2.12)$$

and since the bracketed quantities are constant in time, let

$$E_0 \cos \alpha = E_{01} \cos \alpha_1 + E_{02} \cos \alpha_2 \quad (2.13)$$

and

$$E_0 \sin \alpha = E_{01} \sin \alpha_1 + E_{02} \sin \alpha_2 \quad (2.14)$$

The substitution above is not obvious, but it is legitimate as long as E_0 and α can be solved for. To do so, square and add Equations 2.13 and 2.14;

$$E_0^2 = E_{01}^2 + E_{02}^2 + 2E_{01}E_{02} \cos(\alpha_2 - \alpha_1) \quad (2.15)$$

and divide Equation 2.14 by 2.13 to get

$$\tan \alpha = \frac{E_{01} \sin \alpha_1 + E_{02} \sin \alpha_2}{E_{01} \cos \alpha_1 + E_{02} \cos \alpha_2} \quad (2.16)$$

The total disturbance then becomes;

$$E = E_0 \cos \alpha \sin \omega t + E_0 \sin \alpha \cos \omega t \quad (2.17)$$

or

$$E = E_0 \sin(\omega t + \alpha) \quad (2.18)$$

Thus a single disturbance results from the superposition of the sinusoidal waves E_1 and E_2 . The composite wave is harmonic and of the same frequency as the constituents, although its amplitude and phase are different.

2.1.1.4 Conditions for Interference

Conventional light sources produce light that is a mix of photon wavetrains. At each illuminated point in space there is a net field that oscillates (through roughly a million cycles), for less than 10ns or so, before it changes phase.

This interval over which the lightwave resembles a sinusoid is a measure of what is called its temporal coherence.

As observed from a fixed point in space, the passing lightwave appears fairly sinusoidal for some number of oscillations between abrupt changes of phase. The corresponding spatial extent over which the light wave oscillates in a regular, predictable way is called the coherence length.

To produce a stable interference pattern, the light sources must be coherent and have the same frequency. Until the advent of the laser it was a working principle that no two individual sources could ever produce an observable interference pattern. The coherence time of lasers, however, can be appreciable (of the order of milliseconds), and interference via independent lasers has been detected electronically (though not yet by the rather slow human eye). The most common means of overcoming this problem, as we shall see, is to make one source serve to produce two coherent secondary sources.[17]

2.1.2 Producing Fringe Patterns

The fringe patterns, or interferograms, which are treated in this work are produced by the interference of light. Such patterns are highly useful as they can be used to code a variety of physical measurements.

A divided beam is used to generate an interferogram. One part of the beam is used as a phase reference, and the path of the rest of the beam is influenced in some way. For example this could involve the latter 'object beam', being reflected from an object to record displacement, or passing through a translucent object to record refractive index. It is possible to decode and quantify the measurement parameter by analysis of the fringe pattern formed when the beams are recombined.

The device which controls the interference of the light beams is called an interferometer. The earliest and most widely known interferometer is the Michelson. This was the interferometer used by Michelson and Morley to explore the existence of the luminiferous aether at the end of the 19th century. The Twyman and Green interferometer, Figure 2.3, is a variation of the Michelson.

There now follows a description of the operation of the interferometer, together with examples of its use in detecting faults in a glass plate or measuring surface deformation.

Imagine a plane wave coming from the point source S through the lens O_1 . This wave is divided in two by the beam splitter B .

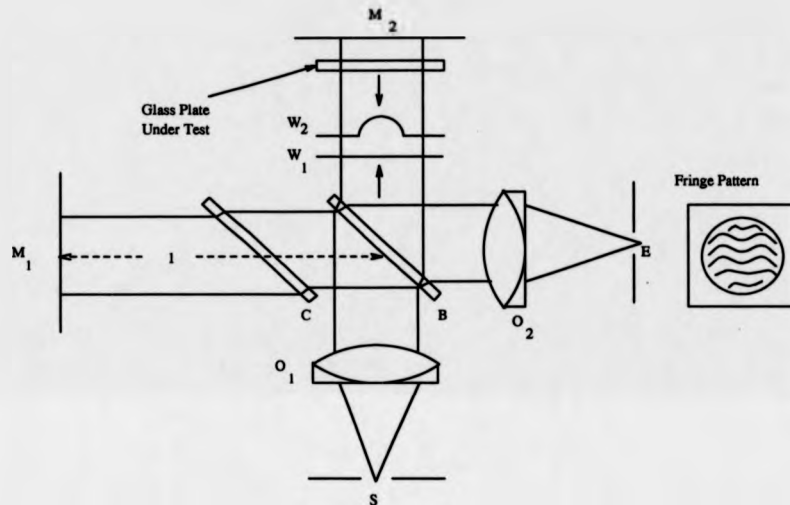


Figure 2.3: Twyman and Green's Interferometer

One of the divided beams is reflected back from the plane mirror M_1 and the other from the mirror M_2 . Either beam may have its phase disturbed during its passage, and in this case, assuming the mirrors are ideal, any disturbance will be due to the glass plate under test.

The beams are reunited and brought to focus by a second lens O_2 . The result may be viewed by placing the eye, or a camera, at E .

The compensator plate C is an exact duplicate of the beam-splitter, with the exception of any silvering or thin film coating. With the compensator in place, any optical path difference arises from the actual path difference. It is positioned at an angle of 45 degrees, so that B and C are parallel to each other.

If the two combining wavefronts are plane, the plate having no defects, the result will be uniform in intensity. If there are defects in the plate, a

There now follows a description of the operation of the interferometer, together with examples of its use in detecting faults in a glass plate or measuring surface deformation.

Imagine a plane wave coming from the point source S through the lens O_1 . This wave is divided in two by the beam splitter B .

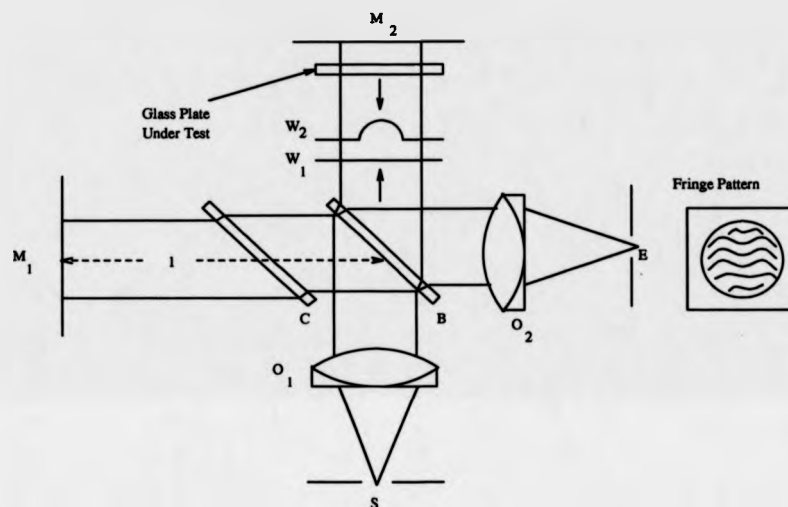


Figure 2.3: Twyman and Green's Interferometer

One of the divided beams is reflected back from the plane mirror M_1 and the other from the mirror M_2 . Either beam may have its phase disturbed during its passage, and in this case, assuming the mirrors are ideal, any disturbance will be due to the glass plate under test.

The beams are reunited and brought to focus by a second lens O_2 . The result may be viewed by placing the eye, or a camera, at E .

The compensator plate C is an exact duplicate of the beam-splitter, with the exception of any silvering or thin film coating. With the compensator in place, any optical path difference arises from the actual path difference. It is positioned at an angle of 45 degrees, so that B and C are parallel to each other.

If the two combining wavefronts are plane, the plate having no defects, the result will be uniform in intensity. If there are defects in the plate, a

deformed wavefront is produced, Figure 2.4 (a), and an interference pattern (fringe pattern, interferogram) is generated, Figure 2.4 (b). The bands in Figure 2.4 (b) are known as fringes. These may be regarded as contour lines of the deformed wavefront. Each fringe corresponds to a different contour of displacement, in the interfering wavefronts, at a spacing of one wavelength. However, elevation is not distinguished from depression. The fringes are sinusoidal in profile. This is examined below.

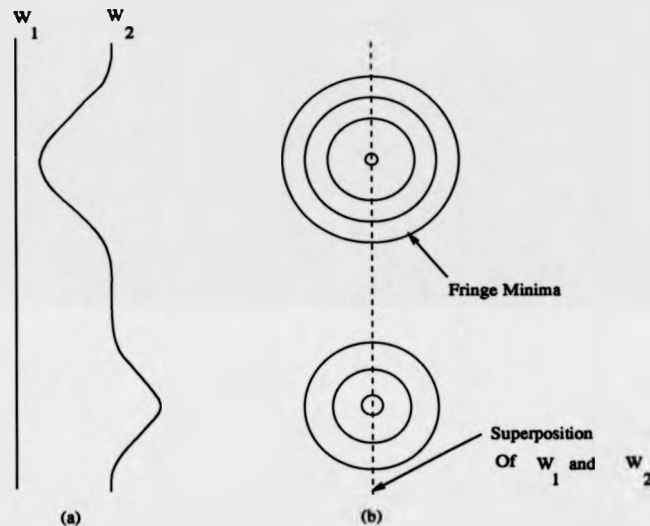


Figure 2.4: The Interference of a Plane and Perturbed Wavefront. (a) The Two Wave Fronts. (b) The Interference Fringe Pattern Generated. (The intensity profile of the fringes generated is sinusoidal)

Suppose that the glass plate is removed. A field of uniform intensity will return. If M_1 is translated towards E , normal to the wavefront, the fringe pattern intensity will vary through maximum and minimum levels in a sinusoidal fashion. If, in this process, M_1 is maintained always parallel to the wavefront, the phase of this cyclic variation will be the same over the entire fringe pattern [19].

Suppose M_1 is returned to its original position and a small area of M_2 is imagined raised. The portion of the interferogram corresponding to the high area will then have an intensity differing from the rest of the pattern. The

exact height of this plateau, with respect to the wavelength of the light used, will determine its intensity in the interferogram. The intensity is related to the number of wavelengths of sinusoidal oscillations through which the high area extends.

If M_1 is again translated, the phase of the cyclic variation of the fringe pattern will no longer be identical over the entire field. That part of the field corresponding to the plateau will go through the cycle with a phase lead relative to the remainder of the pattern. A defect in the plane results in a phase shift of intensity for the corresponding portion of the fringe pattern [19]. Therefore, a fringe contour map of the surface deformation of the mirror or other object is formed.

If the components of the interferometer are assumed to be ideal, the relative reference and test wavefronts are given, respectively by

$$w_1 = A_1 \cos(\omega t + 2kl) \quad (2.19)$$

$$w_2 = A_2 \cos(\omega t + 2kp(x, y)) \quad (2.20)$$

The variables are $k = \frac{2\pi}{\lambda}$, where λ is the wavelength of the light used, l is the pathlength from the beam splitter to the reference surface and where $p(x, y)$ represents the profile of the test surface. The amplitudes of the interfering wavefronts are A_1 and A_2 , respectively.

From the principle of superposition the interference of these two wavefronts generates

$$w = w_1 + w_2 \quad (2.21)$$

The intensity or average power of a light wave is proportional to the square of the amplitude of the wave. So from Equation 2.15;

$$I(x, y, l) \propto A^2 = A_1^2 + A_2^2 + 2A_1A_2 \cos(2k(p(x, y) - l)) \quad (2.22)$$

The interference phenomenon can be used to measure various parameters to the order of the wavelength of light over a small range. The range

of measurement is limited for two reasons. The first being the coherence length of the source and the second being the modulo 2π nature of phase in the interfering beams, which records the measurement. That is, the beams become matched in phase once a wavelength phase difference has passed.

Fringe analysis techniques make it possible to extend the range of measurements by combining the phase information from adjacent fringes into a continuous function. Fringe patterns may be generated under a wide variety of experimental arrangements. Interferometers are not essential, but provide a degree of control over the experiment.

2.1.3 Example From Real Time Holographic Interferometry

An example from Real-time Holographic Interferometry is given below, to measure deformation.

A hologram is made of a specimen object. The hologram is developed and then replaced in the position where it was recorded. If the hologram is then illuminated by the original reference beam a virtual image from the hologram will coincide with the specimen. However, if the object changes shape slightly then two different sets of light waves will reach an observer. The first being from the specimen the second from the holographic image. An observer will then see the reconstructed image covered with a pattern of interference fringes which is, in this case, a contour map of the changes in shape of the object, or deformation [20].

2.1.4 Applications

The interference fringe pattern may represent a number of quantities in addition to the deformation of a surface. It may for example map the amplitude of vibration of a diffusely reflecting surface [20] or be used to measure factors such as strain and stress [21]. Interferometric techniques find application in a wide variety of applications including repetitive calibration, non-destructive testing, and inspection.

Holographic interferometry has been employed in a number of engineering applications, and several examples are given below

- i) It has been developed as a technique for routine use in the evaluation of fan designs for aero engines. It has been used to investigate both the aerodynamic and mechanical behaviour of the rotating fan. Holographic flow visualization provides clear, three-dimensional images of the transonic flow region between the fan blades. Flow features such as shocks, shock/boundary layer interaction, and over-tip leakage vortices can be observed and measured [22, 23, 24].
- ii) There have been many applications in the automotive industry, the most important being in deformation and vibration analyses for the purpose of optimizing design to improve the comfort of ride [25].
- iii) The technique has been applied to verify and adjust tools and machine elements in ultrasonic machining. The main shaft of the machine has a disc to be adjusted. Time average holographic interferometry permits the best diameter of this disc to be found [26].
- iv) Interferometry has been used under water for off shore inspection [27].
- v) Ovrzyn [28] discusses its use in Biomedical Engineering.
- vi) It has been exploited to measure the vibration and deformation fields of cutting tools [29].
- vii) It has been employed in detecting the flaws in composite materials [30, 31]. In the field of composites, Holographic Interferometry is particularly well adapted for the localisation and interpretation of defects such as delaminations, non-adherence and cracks.
- viii) Pryputniewicz discusses the unification of a holographic interferometric method with a finite element method, for the analysis of vibrating beams [32].

- ix) It has been employed to study concrete shrinkage. The deformation of the upper face of a sample, placed in controlled conditions of temperature and humidity, has been observed. Essential parameters for the building of a practical numerical model have then been deduced from the results [33].
- x) It has been used for vibration analysis of such objects as railway bridges, tyres and the rudder units of aircraft [34].
- xi) Sandwich holography has been used in artwork conservation. Its capability for detection of incipient faults or cracks in wooden panel paintings and statues has been tested successfully on models and ancient artifacts under restoration [35].

2.1.5 History Of Fringe Analysis

Interferometric fringe analysis remained for many years a purely manual process. Thomas Young would have been among its first exponents. In the early 1800s he performed experiments to measure the wavelength of light by examining the spacing of interference fringes. It is only very recently that technology has provided a significant aid.

In the past few years computers have increased in power. This in conjunction with a growth in the quality and availability of video equipment, such as low light level cameras, digital frame capture devices and flat bed scanners has stimulated a tremendous growth in the field of image processing.

In the field of fringe analysis these developments have pushed researchers to explore techniques for the automatic analysis of interferograms. The extent to which the process can be automated is a matter of some debate. It has been proposed that expert systems be employed as an aid to automatic methods [36, 1]. Automatic methods aim to eliminate the need for expert knowledge of interferometric methods in the interpretation of fringe patterns and thereby allow it to be used more widely. Automatic techniques are essential if systems are to be used unsupervised, for example in quality assurance on the production line.

2.2 The Fringe Analysis Problem

Many techniques in interferometry generate a two dimensional fringe-contour map of the phase distribution in the form of Eqn. 2.22, which is rewritten in Eqn. 2.23 as it commonly appears when applied to two dimensional interferograms.

$$g(x, y) = a(x, y) + b(x, y) \cos(\phi(x, y)) \quad (2.23)$$

In this equation $a(x, y)$ is the background intensity (which may fluctuate in a real experiment), $b(x, y)$ is the amplitude and $\phi(x, y)$ is the phase.

The fringe analysis software operates over one or more binary coded images. These images contain quantised samples of the intensity values within an interferogram. Unless otherwise stated reference to phase images etc, are with respect to this binary coded form.

Whilst the generation of this kind of fringe pattern permits a direct means of displaying a contour map, it has failings. The sign of the phase cannot be determined, i.e one cannot distinguish between hills and valleys in the map. Accuracy is limited by quantisation of the intensity, non-linearity of the detector, unwanted background variations in $a(x, y)$ and other types of optical and electronic noise.

In order to be widely accepted the results of any experiment in Interferometry, embodied in the interferogram, must be quickly and reliably extracted without expert knowledge. This would best be achieved by the automatic translation of the interferogram into a numerical or pictorial map. The numerical representation would then be easily compared or combined with other kinds of measurement data.

The first obstacle to the extraction of measurement data from a fringe pattern (represented by Eqn. 2.23) is the difficulty in uniting phase measurements from neighbouring fringes. This is necessary to construct a continuous phase map of the measurement parameter across the complete field. The second related impediment, is the problem of distinguishing between elevation and depression.

One method of fringe analysis which is still very popular, and intuitive, is fringe tracking. This is aimed at uniting data from adjacent fringes. The directional ambiguity under this regime is either resolved from shaping the

experimental arrangement so that fringes are generated in an incremental way (by the application of tilt) or by interactive means [37].

However, several, more generally applicable, automatic techniques for the solution of directional ambiguity have been evolved. The most prominent of these are the Fourier Transform Technique (FTT) [6, 7] and Phase Stepping (or Quasi Heterodyning) [2, 3, 4, 5]. In the case of the FTT, a tilt or translation is used to solve the elevation/depression problem by yielding carrier fringes which are interpreted during an FFT process. In the case of Phase Stepping the solution is arrived at by combining several interferograms at different phases. Both techniques yield a 'phase fringe' pattern which contains within it a coding of direction as well as displacement. These techniques are reviewed later in this chapter. This type of fringe pattern is usually referred to as a wrapped phase map or, as it is derived from the tangent of phase, a 'tan' fringe field.

The wrapped phase map contains a coding of elevation and depression, as each point's 'intensity' (when displayed as a grey scale image), is a measure of phase. For each fringe the intensity ranges from black at one extreme, representing a fringe phase of 0, to intense white at the other representing a phase of 2π . Because the phase fringe field records a direct measurement of phase, its analysis is known as Phase Unwrapping. Figure 2.5 shows an example of a computer generated wrapped phase map.

Such fringe fields are not usually analysed by fringe tracking. This technique has been replaced by fringe counting, which involves traversing the scan lines of the digitally processed field in search of the phase roll over points at the fringe edges. These are denoted by a sudden white/black or black/white transition in intensity between adjacent pixels. The aim is that by recording the positions of such edges and the direction of phase roll over, by a suitable edge detection strategy, the phase may be summed in adjacent fringes to produce an unwrapped map. An example of this procedure is given by Nakadate [11] with respect to Speckle images.

In a Holographic system noise represents a problem to such a procedure, as it disrupts the fringe edges, upon which the technique relies. In a Speckle system the fringe field inherently consists of discrete points of information, presenting a much more serious problem. Therefore some other form of

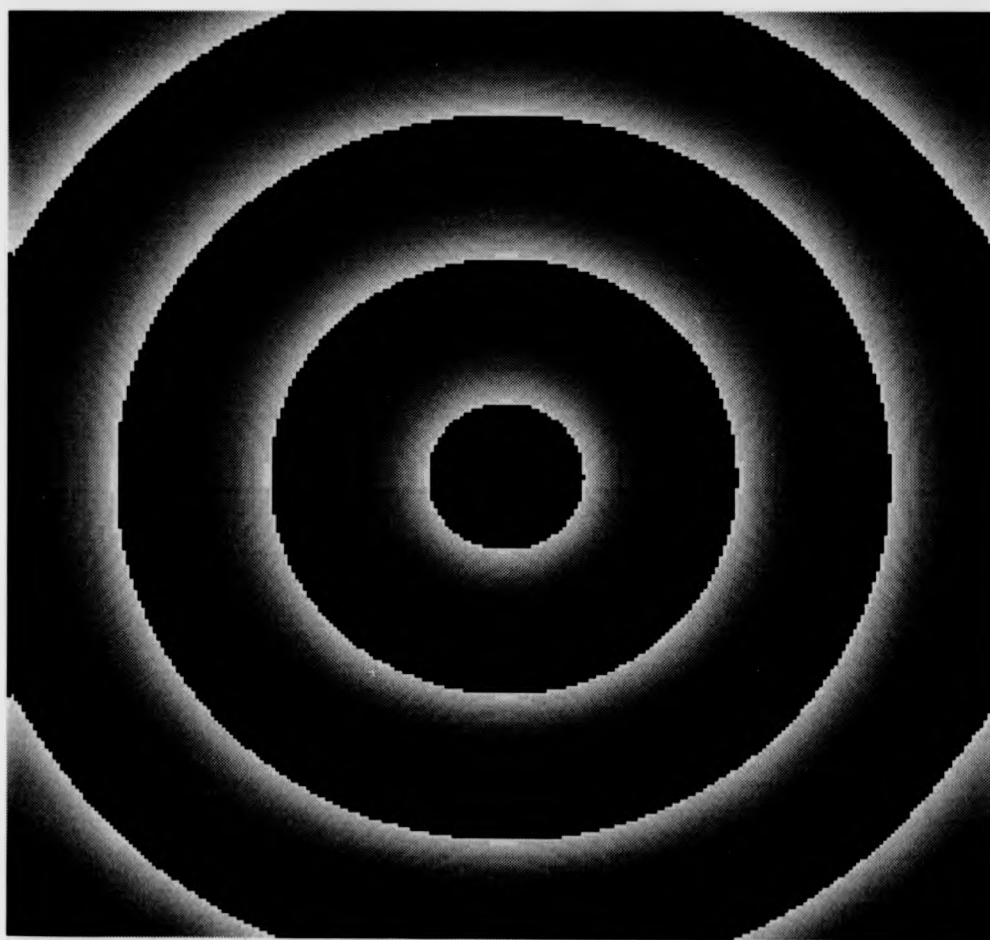


Figure 2.5: Example Wrapped Map

processing must also be applied. The problems of noise and Speckle may be greatly reduced by the use of digital low pass filters, such as an average or median. These pass the relatively low spatial undulations of the fringes and filter out the rapid oscillations characteristic of noise.

However, phase unwrapping techniques have recently been developed which are immune to spike noise [13, 14, 15, 16]. Ghiglia describes the discovery of a cellular automaton which can unwrap *consistent* phase data in n dimensions in a path-independent manner and which can automatically accommodate noise-induced (pointlike) inconsistencies. Although robust for *consistent* phase data the procedure does not address inconsistencies which are large in scale. That is, those which are larger than pixel sized spike noise and so may not be detected by a pixel level inspection. As described, the technique requires a large number of iterations to converge. Goldstein et al. [15] and Huntley [16] have described algorithms based on cuts which are immune to such noise and computationally efficient.

The algorithm described here is robust, immune to noise and computationally efficient. However, the advantage of the MSTT algorithm over the former algorithms is that it is scalable. Utilizing small 'tiles' it may unwrap a consistent phase map containing noise in a related manner to the methods of Ghiglia, Goldstein and Huntley. By increasing the size of tiles, larger local features of inconsistency may be detected. Ghiglia describes these as 'natural or aliasing-induced path inconsistencies'.

Natural path-inconsistencies can occur for a variety of reasons, because two independent objects overlap, for example. In satellite interferometry the tops of mountains and hills can be closer to the radar than their bases, so that in the interferograms, they appear to lean toward the radar. This is known as the layover effect. Shadows can cause large discontinuities, as can spaces between objects or their parts. Such spaces are especially problematic when they occur with a spatial frequency of the same order as the fringe spacing, as their size means that they may not be filtered out.

Aliasing-induced inconsistencies can be generated by the device recording the fringe pattern. This occurs when the device does not have a sufficient bandwidth to cope with that of the fringe field. That is, the fringes are less than the size of a pixel in width.

In general, it is impossible to produce a totally unambiguous solution to a phase map with 'natural or aliasing induced path inconsistencies'. However, the MSTT technique incorporates strategies which minimise their effects where possible. In order to do so the strategy measures factors such as local fringe density, low modulation noise, fringe edge terminations and the extent to which the solution of neighbouring field areas agree.

The new MSTT method has an inherently high tolerance of errors in the interferogram, and operates on the 'phase fringe' fields generated by either the Quasi-Heterodyne or FFT methods.

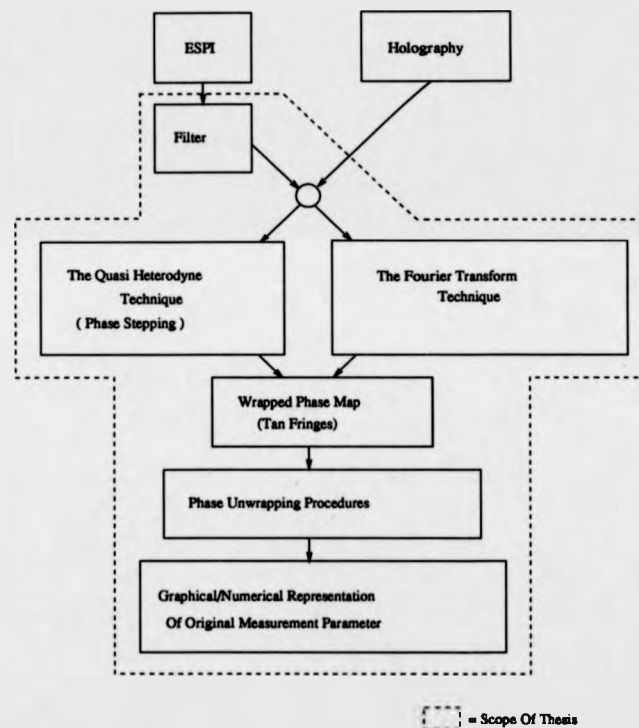


Figure 2.6: Relationship of Elements in Sector of Fringe Analysis Considered

The interaction of the various elements in a fully automated fringe analysis system are shown in Figure 2.6. The two systems, leading to the generation of a wrapped phase map, allow some flexibility. It may be that analysis from a single interferogram is essential, in which case the Fourier Transform

Technique may be employed, alternatively a need for greater accuracy [38] may promote the use of Quasi Heterodyning, which requires more than one interferogram.

2.3 The Analysis Of A Conventional Interferogram By Fringe Tracking

A first approach in the analysis of an intensity fringe pattern might be fringe tracking. This involves identifying the fringe positions and following their track across the interferogram. The technique can be applied to a range of applications.

As G.T Reid [5] outlines, a number of fringe tracking methods have been proposed [8, 9, 10]. Although these methods differ in detail, they usually rely on the following procedure:

- i) Filter the image. Chambless [39] describes the application of the FFT to low pass filter speckle images.
- ii) Either (a) fit curves to the intensity data with a view to interpolating between fringe centres [40] or (b) identify and track the intensity maxima and/or minima with a view to skeletonising the pattern and thereby minimising the amount of data which must subsequently be processed [41].
- iii) Number the fringes either interactively [37] or automatically [42].
- iv) Calculate the measurement parameter from the fringe pattern data.

In the analysis of high contrast, low noise interferograms which are produced in many applications of classical interferometry, especially Holographic Interferometry, it is often possible to proceed directly to the fringe tracking or curve fitting stage without filtering the image.

Many other types of interferogram, however, contain comparatively high levels of noise and/or exhibit quite severe variations in fringe contrast and therefore require pre-processing.

Button et al. [8] describe two rudimentary methods of identifying fringe positions in Speckle patterns, based on fringe tracking. The first of these methods requires the manual input of an initial fringe position from which the average intensity along the direction of the fringe track for a given distance is found. The average intensities in two directions on either side of this are also computed. The direction with the minimum value is chosen to define a new point on the fringe and the procedure is repeated. In this way, the fringe is tracked until some predefined condition is satisfied. The second algorithm employs thresholding to reduce the fringes to a binary image. Each fringe then has a single intensity level which is tracked as before after the application of a procedure aimed at removing single pixels of noise. It was noted that the first algorithm occasionally wandered off a fringe track.

The methods described by Nakadate et al. [10] rely to a great extent on the operator indicating the positions of fringe centres with a light pen. Funnell [9] advocates the combination of fringe tracking procedures with a variety of user interactions. The user is permitted to specify a boundary within which to perform the analysis. The fringe centres are determined by selecting points along a line cutting all the fringes whose endpoints are specified by the user. The user may request a low pass filter to help with fringe breaks caused by noise. After each fringe is traced the user also has the option of accepting the trace or making the system try again from another point.

2.3.1 Problems Associated With Fringe Tracking In Completing The Interferogram Analysis

The relative displacement of the tracked fringes, with respect to their neighbours, must then be determined. This is known as assigning fringe order numbers. If the operator has sufficient knowledge of the interferogram then the fringes can be numbered interactively. This procedure might consist of the operator identifying a fringe with a light pen or cursor and then typing the fringe number into the computer. Interactive fringe numbering can be quite hazardous when complicated interferograms are under analysis.

An erroneously numbered fringe can lead to substantial errors in the

final calculation of the measurement parameter. In many circumstances, it is possible to overcome the fringe numbering problem by introducing a substantial degree of tilt [41] to the interferogram so that, the fringes become almost parallel with the greatest gradient in the measurement parameter, and is then seen as a deviation from straightness of the fringes. In this case, the fringe number increases by unity as one moves from one fringe to the next and fringe numbering can be carried out automatically.

The techniques applied by Yatagai, which employ tilt, [43] appear to perform well for the application they considered, an automatic flatness tester for VLSI circuit wafers. The combination of a variety of image processing procedures leads to a correct automatic analysis, contrast enhancement, conversion to a binary image, and thinning. However, the application is a special case. Circuit wafers are not likely to have holes in them, for example. Yatagai uses a region labeling function of his image processing hardware to distinguish between fringes. If a gap were to exist between sections of the same fringe, such as would be produced by a hole, the sections would probably be labeled as separate fringes, and an error introduced.

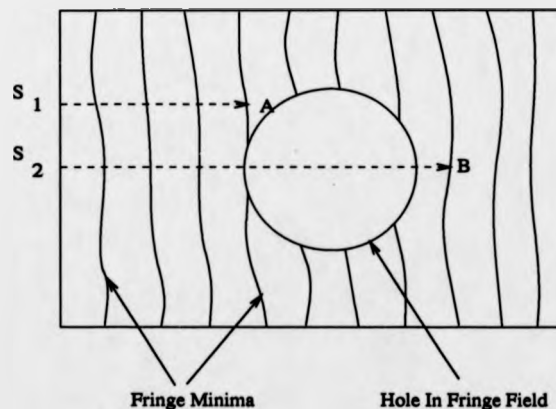


Figure 2.7: Example Problem in the Determination of Fringe Orders in a Discontinuous Field

Under the tilt regime, a discontinuous field makes it difficult to automatically determine fringe orders. Referring to Figure 2.7, the fringe at position *A* could easily be given the same order as that at position *B*, simply de-

pending on the scan used to determine the order numbers. S_1 and S_2 are two example scans. In some applications it may also be difficult to introduce sufficient tilt [5] to encode the underlying signal, this is a problem in common with the FTT method of fringe pattern analysis.

Fringe tracking is not an automatic system for the analysis of interferograms, except under very specific conditions.

It becomes necessary to identify clearly two important properties that a successful automatic fringe analysis system must embody.

- i) The solution of the elevation/depression problem in the widest possible range of circumstances.
- ii) The correct identification of fringe orders.

To acquire the second property it seems evident that an algorithm must certainly contain some mechanism for evaluating the quality of data in the field of the interferogram. Without such information a system could produce a confused result, and be incapable of giving any automatic indication of where a problem may have occurred. It is difficult to see how the problems of a fringe following system, e.g. tracking being halted at a fringe break or wandering over to another fringe, may be resolved without human intervention.

2.4 Electronic And Quasi Heterodyning

2.4.1 Electronic Heterodyning

The first part of this next section is based largely on the review paper by Reid [5].

The heterodyning technique is employed to resolve directional ambiguity, between hills and valleys. It involves the use of a special type of interferometer. Although more complex than conventional interferometers, such as the Twyman and Green, these are constructed around the needs of automatic analysis rather than visual interpretation. This means that they do not possess the same weaknesses as classical interferometers, i.e. sensitivity to background variation [5].

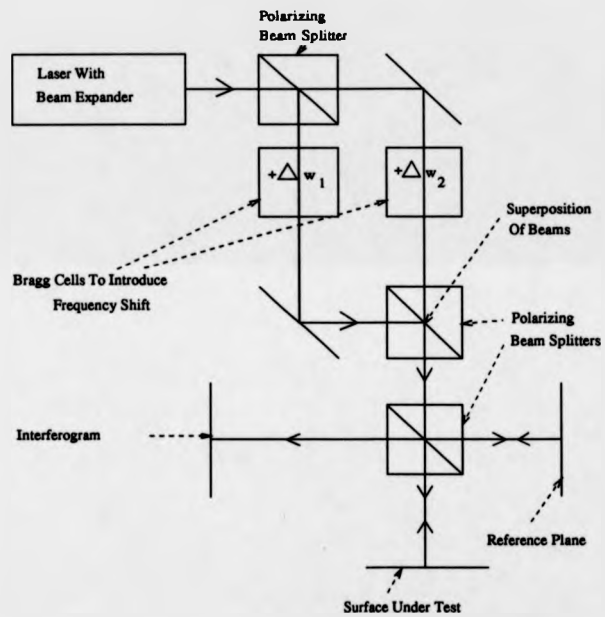


Figure 2.8: Heterodyned Twyman and Green Interferometer

In principle, the heterodyne interferometer consists of a conventional two-beam interferometer in which the fringe pattern is analysed electronically. The interferometer may be of any type having two separate beam paths. In one path the beam either passes through or is reflected from the optical element being tested, while in the other, the original wavefront is preserved as a reference plane [19]. A heterodyned Twyman and Green interferometer is shown in Figure 2.8 [5]. Bragg cells are placed in both beams of the interferometer to impose unequal frequency shifts, $\Delta\omega_1$ and $\Delta\omega_2$, on the interfering laser beams. Denoting ω as the optical frequency of the light leaving the laser, E as the optical field amplitude and ϕ as the optical phase, we can represent the two interfering laser beams, at time t , by

$$E_1 = E_{01} \cos((\omega + \Delta\omega_1)t + \phi_1) \quad (2.24)$$

$$E_2 = E_{02} \cos((\omega + \Delta\omega_2)t + \phi_2) \quad (2.25)$$

A photoelectric detector, lying within the interference field, will generate a photocurrent, i , of the form

$$i = \langle E_1 + E_2 \rangle^2 \quad (2.26)$$

where $\langle \rangle$ represents time averaging due to the finite bandwidth of the detector. Expanding Eqn. 2.26, we find

$$\begin{aligned} i = & E_{01}^2 \langle \cos^2((\omega + \Delta\omega_1)t + \phi_1) \rangle + \\ & E_{02}^2 \langle \cos^2((\omega + \Delta\omega_2)t + \phi_2) \rangle + \\ & E_{01}E_{02} \langle \cos((2\omega + \Delta\omega_1 + \Delta\omega_2)t + \phi_1 + \phi_2) \rangle + \\ & E_{01}E_{02} \langle \cos((\Delta\omega_1 - \Delta\omega_2)t + \phi_1 - \phi_2) \rangle \end{aligned} \quad (2.27)$$

Assuming that the bandwidth of the detector exceeds $(\Delta\omega_1 - \Delta\omega_2)$ but is much less than ω , then only the last term of Eqn. 2.27 will contribute to the alternating photocurrent. Under this condition, therefore, the phase $(\phi_1 - \phi_2)$ of the alternating photocurrent which is generated at a given pixel is equal to the phase of the interferogram at that pixel.

The use of heterodyne interferometry allows certain issues for the fringe analysis software to be eliminated. The phase values produced are not affected to a great extent by stationary noise and variations in contrast of the fringes. The greatest benefit is that the increase or decrease of phase with an increasing or decreasing fringe order number means that the complexities of determining fringe orders are reduced. However, noise and the possibility of a non contiguous fringe field are still problems.

2.4.2 Quasi-Heterodyning

The Heterodyne method relies on the detector being sensitive to the difference in frequencies between $\Delta\omega_1$ and $\Delta\omega_2$, and not sensitive to frequencies above ω . This sets a specification for any photo detector that might be used.

Ideally an array of detectors is needed, so that phase can be sampled at a large number of positions. At first it might be thought that video type cameras, especially Charge Coupled Devices which typically have a usable resolution of 512 by 512 detectors, would be quite suitable. However the capture frequency of such cameras is limited to video rates, around 25Hz. This is incompatible with the frequency shifting techniques available which give $(\Delta\omega_1 - \Delta\omega_2)$ in the range of kilohertz.

The Quasi-Heterodyne technique [2, 3, 4, 5] overcomes this problem by sampling at discrete steps of phase and time. It is probably the most important technique that has been developed for the automatic determination of elevation and depression. It is compatible with video systems and like the Heterodyne technique is tolerant of local intensity variations.

Figure 2.9 shows the arrangement of the interferometer used in Quasi-Heterodyning. Note the addition of the piezoelectric translator to alter the pathlength l , for the different phase positions. Referring back to the discussion at the beginning of the chapter and Equations 2.19, 2.20 and 2.22.

$$w_1 = A_1 \cos(\omega t + 2kl)$$

$$w_2 = A_2 \cos(\omega t + 2kp(x, y))$$

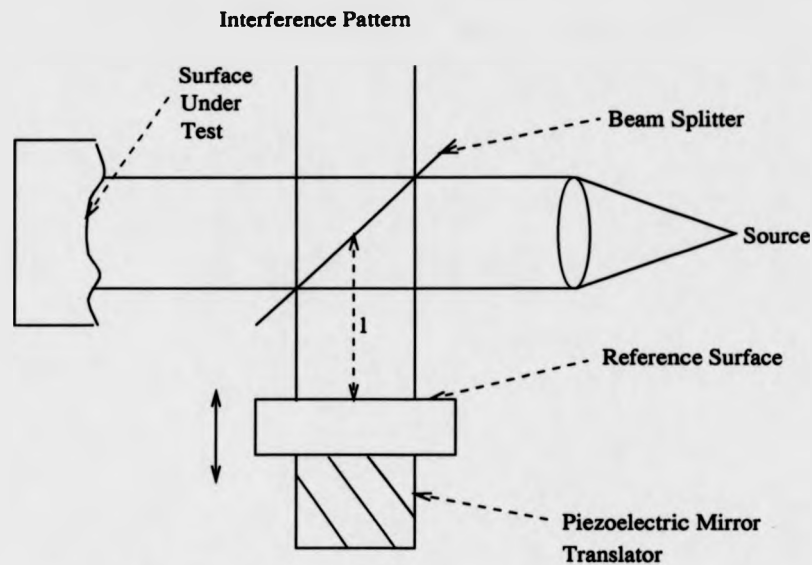


Figure 2.9: Interferometer Used in Quasi-Heterodyning

with

$$I(x, y, l) \propto A^2 = A_1^2 + A_2^2 + 2A_1A_2 \cos(2k(p(x, y) - l))$$

It may be seen that the real factor of interest is now the profile of the test surface $p(x, y)$.

Eqn. 2.22 may be rewritten using the identity

$$\cos(A - B) = \cos A \cos B + \sin A \sin B \quad (2.28)$$

to give Eqn. 2.29 shown below

$$I(x, y, l) \propto a_0 + a_1 \cos 2kl + b_1 \sin 2kl \quad (2.29)$$

In Eqn. 2.29 the variables are defined as follows

$$a_0 = A_1^2 + A_2^2 \quad (2.30)$$

and the cosine coefficients a_1 and b_1 , which are functions of x and y are

$$a_1 = 2A_1A_2 \cos 2kp(x, y) \quad (2.31)$$

and

$$b_1 = 2A_1A_2 \sin 2kp(x, y) \quad (2.32)$$

The aim is to find the profile of the test object. First of all it is noted that by obtaining values for a_1 and b_1 the profile may be calculated from;

$$2kp(x, y) = \arctan\left(\frac{b_1}{a_1}\right) \bmod 2\pi \quad (2.33)$$

where 'mod 2π ' indicates that the map obtained is wrapped every 2π phase change of I .

The profile can therefore be found from I provided that a_1 and b_1 can be isolated. I is obtained from discrete samples at varying l using a detector sampling in x and y . This might be achieved, for example, by using a CCD camera on a traverse. Such samples represent a sequence of fringe patterns captured at different phases of I .

Note, the proportionality of I in Equation 2.29. Any constant of proportionality between the sampled intensity values and I is cancelled out in the division of b_1 by a_1 in Equation 2.33 when actually calculating the profile. However, a non-linear response in the detector, clearly, has an effect on the measurement. Linearisation of the response of the detector is a subject which has received little comment in the literature.

It is now shown how a_0 , a_1 and b_1 can be obtained from I . The continuous case is considered first. Suppose i is equal to the right hand side of Equation 2.29;

$$i(x, y, l) = a_0 + a_1 \cos 2kl + b_1 \sin 2kl \quad (2.34)$$

a_0 is obtained by integrating over one cycle of length L , for constant x and y . As a result of the integration this gives a_0L . The rest of the expression becomes zero as the integrals of sin and cos over one cycle are zero. So a_0 is given by,

$$a_0 = \frac{1}{L} \int_{l_0}^{l_0+L} i(x, y, l) dl \quad (2.35)$$

The coefficient a_1 is obtained by multiplying both sides of Equation 2.34 by $\cos 2kl$ and again integrating over one cycle. The orthogonality of the sine and cosine functions means that the sine part of $i(x, y, l)$, that is $b_1 \sin 2kl$ is eliminated in the integration, whilst the cosine part becomes $\cos^2 2kl$. The integral of \cos^2 over one period is a half, hence the $\frac{2}{L}$ in the equation below,

$$a_1 = \frac{2}{L} \int_{l_0}^{l_0+L} i(x, y, l) \cos 2kl dl \quad (2.36)$$

Similarly, an expression for b_1 may be obtained by multiplying by $\sin 2kl$ and integrating over one cycle,

$$b_1 = \frac{2}{L} \int_{l_0}^{l_0+L} i(x, y, l) \sin 2kl dl \quad (2.37)$$

For a discussion of Orthogonal Functions and the Trigonometric Fourier Series, upon which this process is based, see reference [44].

The next step is to convert this continuous representation to one that applies to discrete samples. Below, the integral of intensity over one cycle is approximated by a discrete sum,

$$\int_{l_0}^{l_0+L} i(x, y, l) dl \approx \sum_{j=0}^{N-1} i(x, y, l_j) \Delta l = \Delta l \sum_{j=0}^{N-1} i(x, y, l_j) \quad (2.38)$$

From this description the following discrete representations for a_0 , a_1 and b_1 are obtained,

$$a_0 = \frac{\Delta l}{L} \sum_{j=0}^{N-1} i(x, y, l_j) = A_1^2 + A_2^2 \quad (2.39)$$

the coefficients a_1 and b_1 are given by;

$$a_1 = \frac{2\Delta l}{L} \sum_{j=0}^{N-1} i(x, y, l_j) \cos 2kl_j = 2A_1 A_2 \cos 2kp(x, y) \quad (2.40)$$

$$b_1 = \frac{2\Delta l}{L} \sum_{j=0}^{N-1} i(x, y, l_j) \sin 2kl_j = 2A_1 A_2 \sin 2kp(x, y) \quad (2.41)$$

where N is the number of intensity samples. The phase at the given x, y

sample position $p(x, y)$ is then, as was seen earlier, given by

$$2kp(x, y) = \arctan\left(\frac{b_1}{a_1}\right) \bmod 2\pi \quad (2.42)$$

Equations 2.40 and 2.41 are a general form of the Phase Stepping equations. It is often the case that just 3 or 4 fringe fields are combined, at phase steps of either 90 or 120 degrees. In the case of three fringe fields at a phase step of α , the phase ϕ at a given pixel may be calculated more concisely from

$$\phi = \arctan \left[\frac{(I_3 - I_2) \cos \alpha + (I_1 - I_3) \cos 2\alpha + (I_2 - I_1) \cos 3\alpha}{(I_3 - I_2) \sin \alpha + (I_1 - I_3) \sin 2\alpha + (I_2 - I_1) \sin 3\alpha} \right] \quad (2.43)$$

where I_1 , I_2 and I_3 are the intensities of the interferograms at the three phase positions α , 2α and 3α respectively [2].

2.5 The Fourier Transform Technique

Takeda et al. [6] describe a method of extracting phase values from a single interferogram by relying on the addition of carrier fringes. These carrier fringes may be introduced by a tilt or translation of the object or part of the optical set up. In fact, this is a similar process to that required for the reliable analysis of an interferogram by fringe tracking. No closed loop, or split fringes should be present.

The direction in which the tilt on the object has been applied, to obtain the carrier fringes, is not recoverable from the interferogram. This may lead to the wedge shape in the measurement parameter (which the carriers represent) being reversed, so that its slope becomes normal to its original slope, in which case hills become valleys and valleys become hills. However, the direction in which this wedge is inclined may usually be supplied as an input to the analysis software and does not represent a serious handicap.

Figure 2.10 shows a computer generated example of an interferogram. In a real experiment such an interferogram might, for example, be generated by a double exposure holographic method. Such an interferogram could represent a circular plate with pressure applied at its centre, which has caused a small deformation in the range of micrometres. The object is not tilted

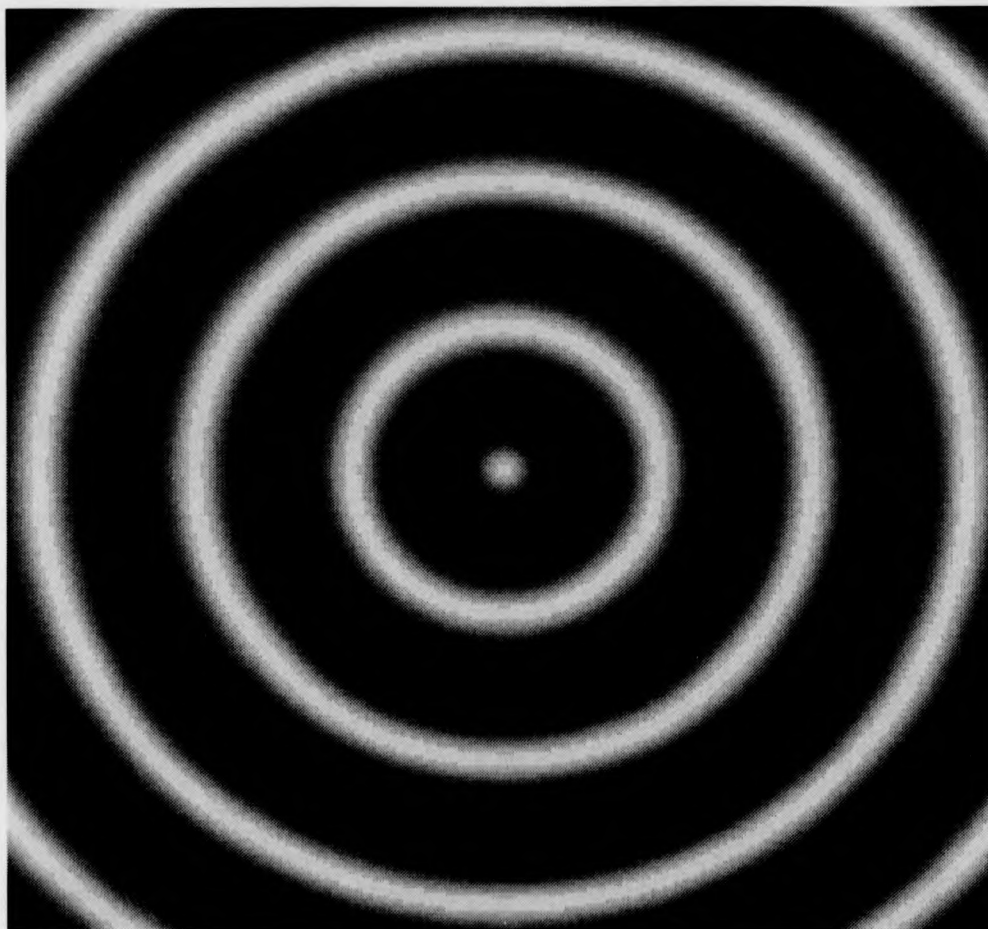


Figure 2.10: Computer Generated Example of Interferogram (The Subject Simulated is a Circular Disc with Pressure Applied at its Centre to Induce Deformation)

and so there are no carrier fringes involved.

Figure 2.11 shows another computer generated interferogram. This simulates the interferogram of a perfectly flat plate, tilted very slightly until it forms a wedge. In effect the image shows a set of unmodulated carrier fringes, due to the flatness of the plate.

Figure 2.12 shows the interferogram that would result from applying a central deformation to a flat plate and tilting it, between exposures. In this case the carrier fringes are modulated by the deformation of the plate. Figure 2.5 shows the wrapped phase map produced by an FFT analysis.

Figure 2.13 shows a real interferogram of a metal disc made in this way.

An interferogram containing carrier fringes may be represented by the equation below,

$$g(x, y) = a(x, y) + b(x, y) \cos(2\pi f_0 x + \phi(x, y)) \quad (2.44)$$

where $a(x, y)$ is the background intensity, $b(x, y)$ is the amplitude, $\phi(x, y)$ is the phase and f_0 is the spatial-carrier frequency.

The bandwidth of the image capture device employed should be sufficient to satisfy the sampling theory requirements of Nyquist for the spatial-carrier frequency f_0 . The carrier fringes run horizontally, so this is of particular importance in the x direction.

The input fringe pattern is rewritten in the following form for convenience of explanation:

$$g(x, y) = a(x, y) + c(x, y) \exp(2\pi i f_0 x) + c^*(x, y) \exp(-2\pi i f_0 x) \quad (2.45)$$

with

$$c(x, y) = \frac{b(x, y)}{2} \exp[i\phi(x, y)] \quad (2.46)$$

where $*$ denotes a complex conjugate.

Next, Eqn. 2.45 is Fourier transformed with respect to x by the use of a set of one dimensional Fast Fourier Transforms, one for each scan line, which gives

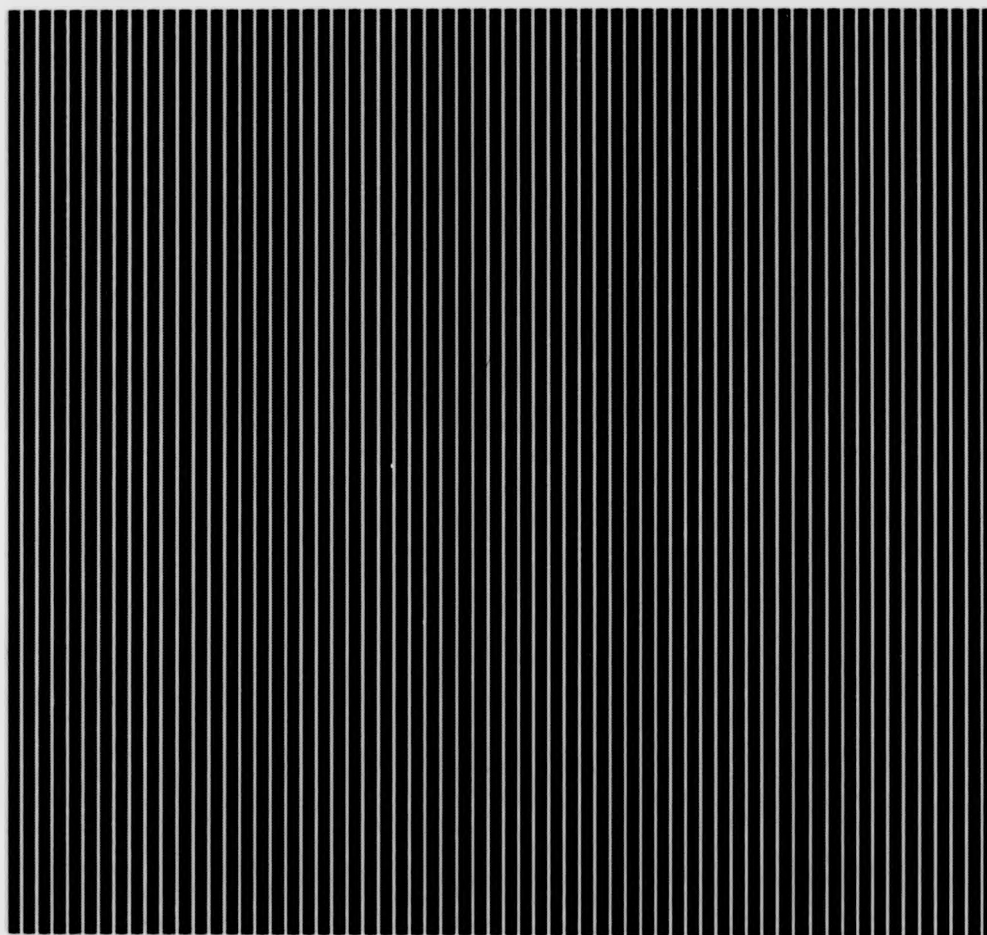


Figure 2.11: Computer Generated Example of Interferogram (The Subject Simulated is a Tilted Flat Plate)

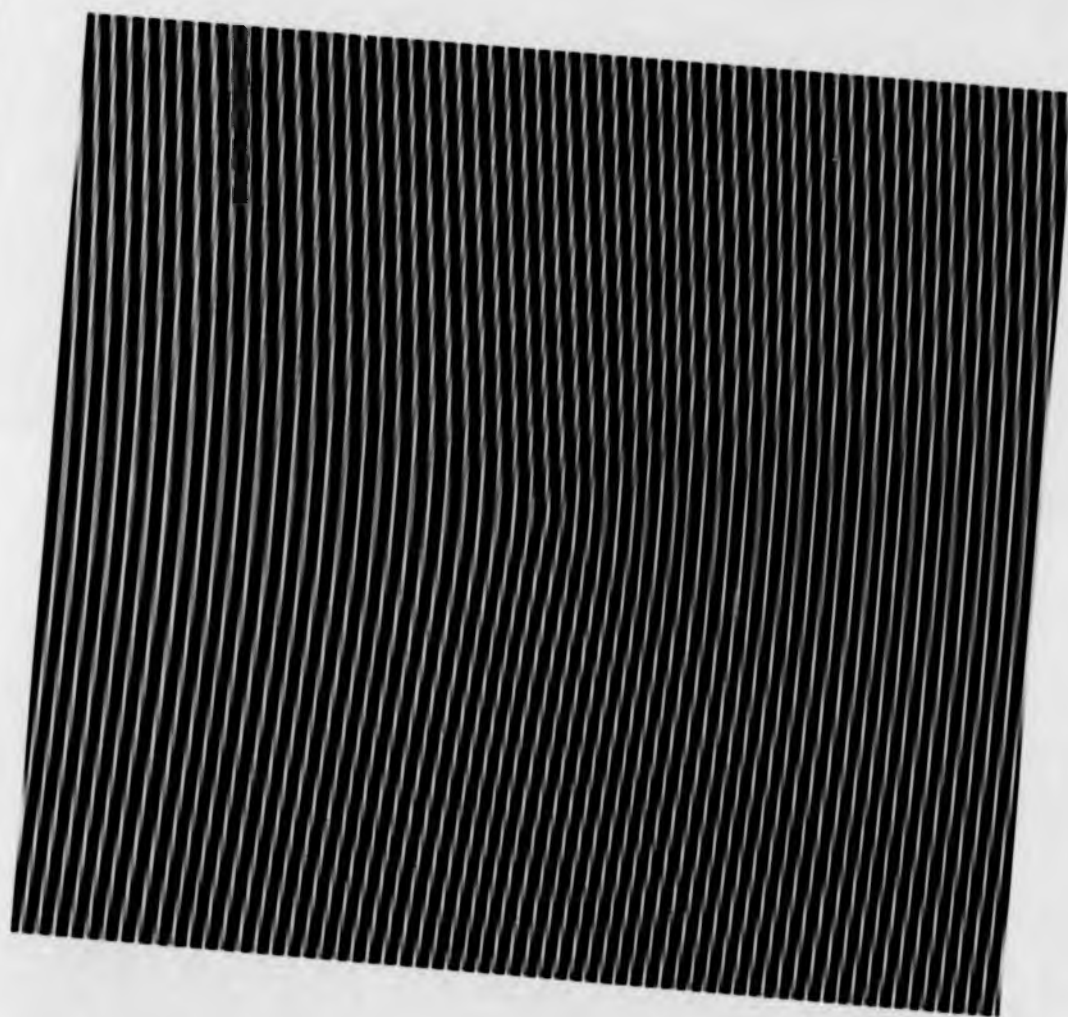


Figure 2.12: Computer Generated Example of Interferogram (The Subject Simulated is a Centrally Deformed Plate which has been Tilted between Exposures)

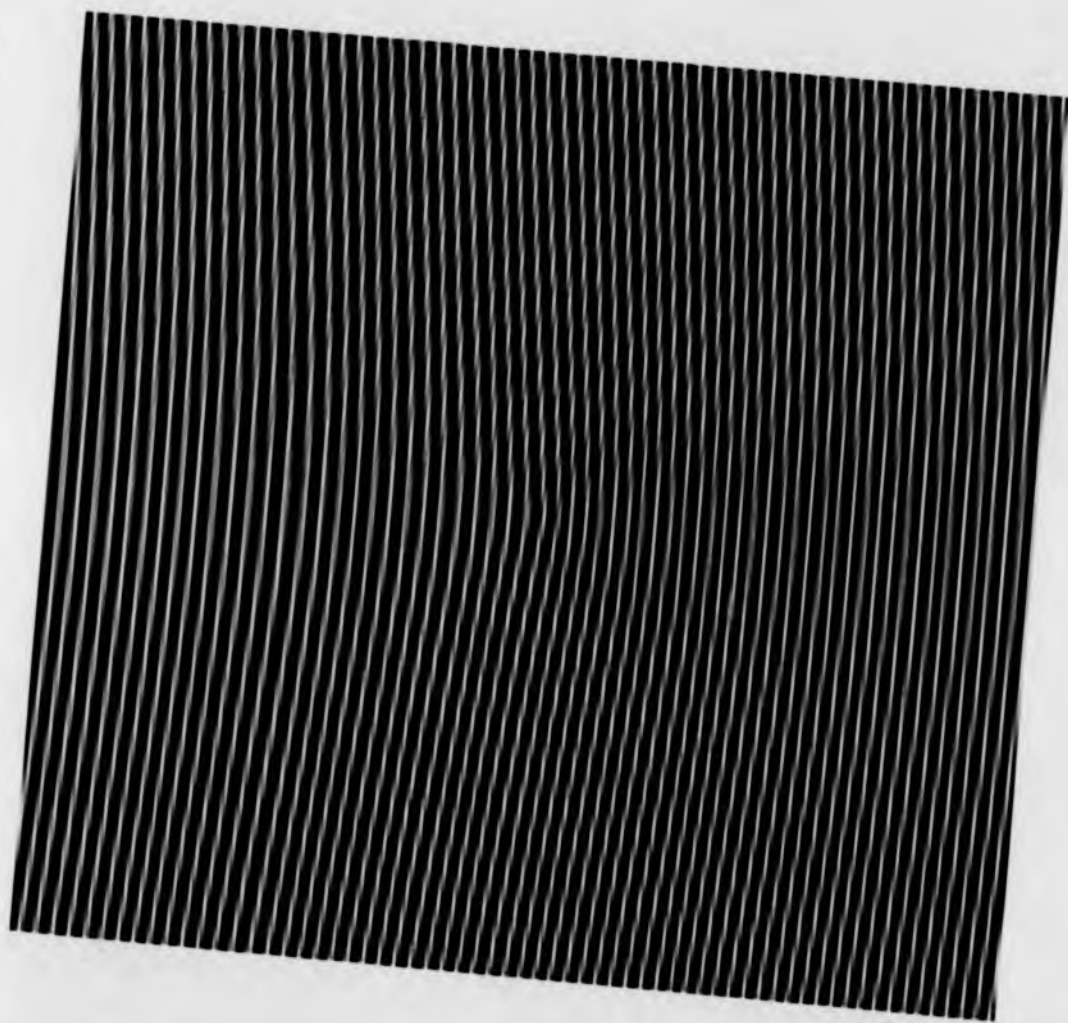


Figure 2.12: Computer Generated Example of Interferogram (The Subject Simulated is a Centrally Deformed Plate which has been Tilted between Exposures)

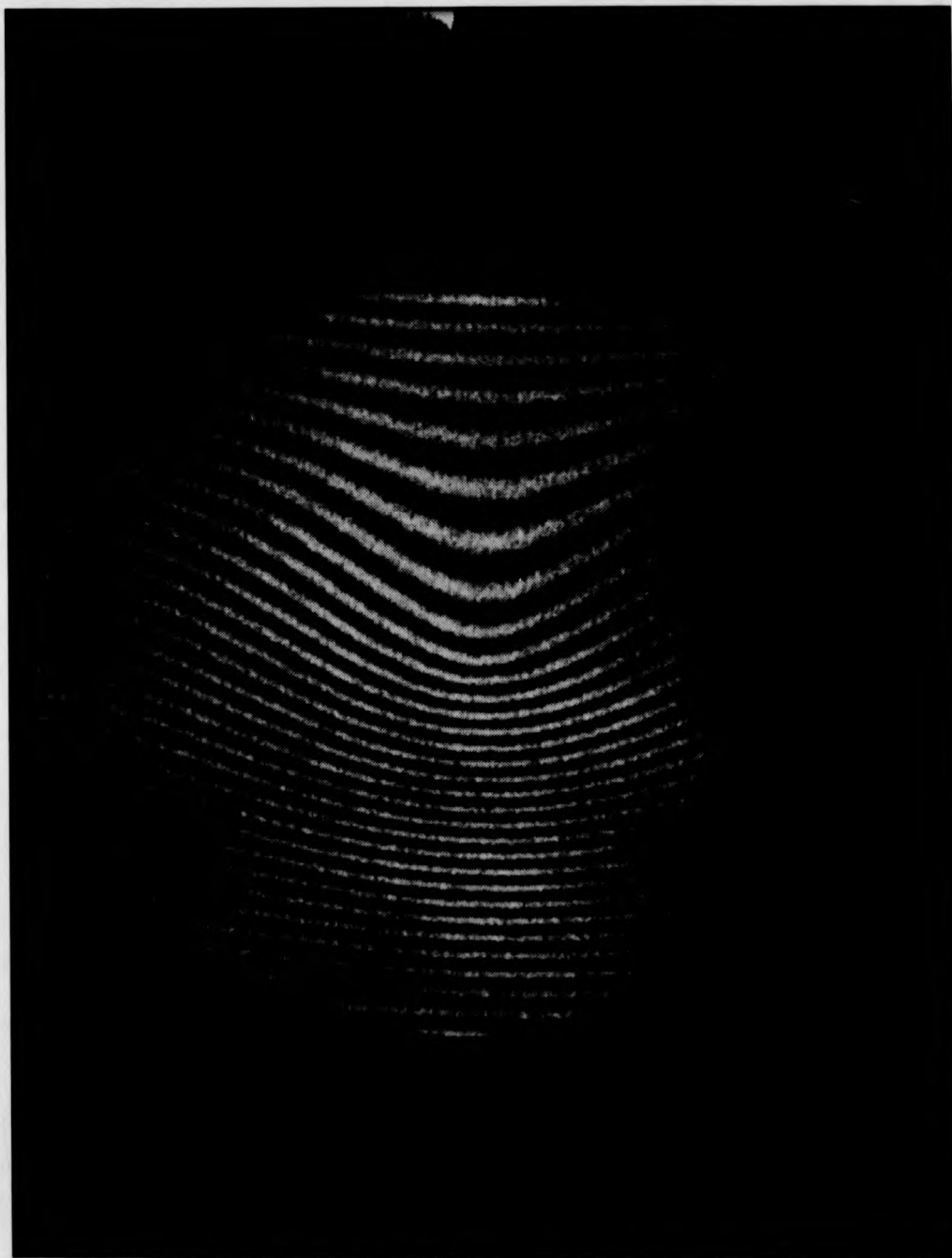


Figure 2.13: Real Holographic Interferogram of Centrally Deformed Plate

$$G(f, y) = A(f, y) + C(f - f_0, y) + C^*(f + f_0, y) \quad (2.47)$$

where the capital letters denote the Fourier spectra and f is the spatial frequency in the x direction.

Assuming $a(x, y)$, $b(x, y)$ and $\phi(x, y)$ have frequencies which are much lower than the spatial carrier frequency f_0 , then their spectra will be separated as shown in Figure 2.14. In this figure the y axis is normal to the page.

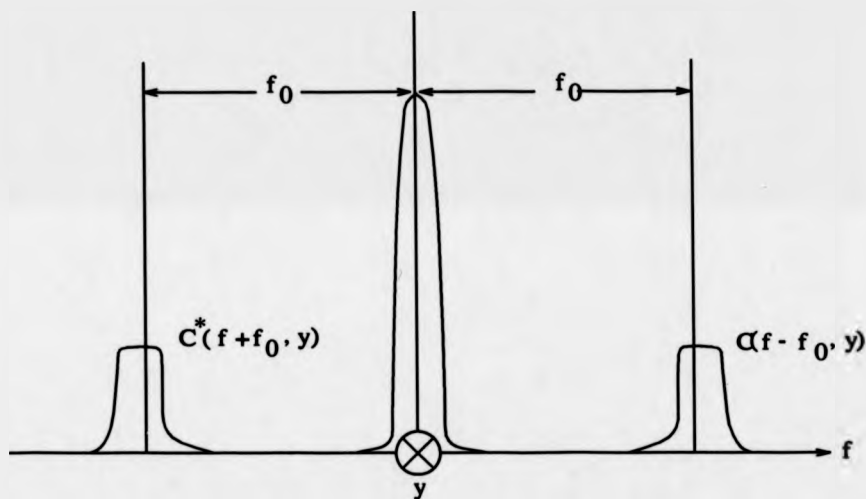


Figure 2.14: Separated Spectra of Fringe Pattern

The algorithm utilises one of the two side lobes. Suppose $C(f - f_0, y)$ is chosen. Womack's Complex Exponential Window Algorithm [45] is identical to Takeda's. A translated Hamming function is suggested, as a filter window, to extract the desired side lobe. An ideal filter would cut away the zero peak, stemming from the slowly varying background illumination, and filter out high frequency speckle noise. The spectrum would then be translated along the frequency axis by f_0 so that it sits in the position shown in Figure 2.15.

Taking the inverse Fourier transform of $C(f, y)$ with respect to x , we find $c(x, y)$. The phase may then be calculated from Eqn. 2.50 below

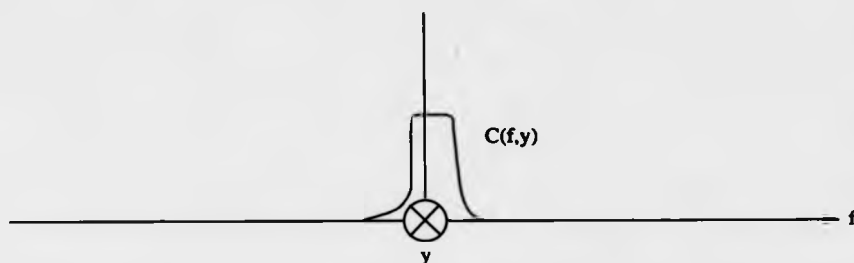


Figure 2.15: One of the Side Lobes Translated to the Origin

$$c(x, y) = \frac{b(x, y)}{2} \exp[i\phi(x, y)] \quad (2.48)$$

Recall

$$(\exp[i\phi(x, y)] = \cos \phi(x, y) + i \sin \phi(x, y)) \quad (2.49)$$

$$\phi(x, y) = \arctan \left[\frac{\Im[c(x, y)]}{\Re[c(x, y)]} \right] \quad (2.50)$$

in which $\Re[c(x, y)]$ and $\Im[c(x, y)]$ represent the real and imaginary parts of $c(x, y)$, respectively.

The Fourier transform method as described here performs a one dimensional FFT on each scan line of the field. It utilises each sample point in the computation of the frequency spectrum and so, in the inverse computation which yields phase, every point on a given scan contributes to the phase value given for any individual point. The one dimensional version of the algorithm requires that the carrier fringes are incremental along the horizontal axis of the frame. If the carrier fringes are not horizontal, (perhaps they are diagonal for example) then the algorithm operates on the the horizontal component of the carrier and leaves a ramp in the solution running from the top to bottom of the frame, the gradient of which corresponds to the vertical component of the carrier fringe pattern.

Macy [46] describes a modification to the above technique which utilises a two dimensional FFT. This does not require the carrier fringes to run horizontally.

Kreis [38] later uses a two dimensional transform in the evaluation of an interferogram of a thermally loaded panel. In this example, a hologram is made of an illuminated panel. The hologram is developed and replaced in its original position. A video frame is then captured from this reconstruction. After a temperature change, interference fringes are formed between the hologram and the thermally distorted panel. A video frame of this image is captured.

The frequency spectra of the two video frames, before and after, are computed via a two dimensional FFT. The complex spectrum, yielded by the FFT, of the background is subtracted from the spectrum of the interferogram. This effectively removes the $A(f, y)$ term.

The isolation of a single side lobe is then simply a matter of zeroing one half of the complex spectrum. The method then follows the same lines as before; translation (in a 2D sense), inversion and phase calculation from the arctangent of the ratio of the imaginary and real parts of the inverted FFT.

The procedure described by Kreis enables the side lobe to be isolated without a windowing operation in the Fourier domain. That is, the side lobe may be isolated without, necessarily, having to use a window to isolate it from the DC and $A(f, y)$ term. However, this does mean that two images are required. A background image is required in addition to the encoded fringe pattern.

Windowing methods are prone to error, especially if the interferogram contains masked areas, such as the model in a flow experiment (this case is considered in Chapter 5). The $A(f, y)$ term then not only records slight intensity variations across the image, but also the Fourier transform of the masked area (model). The frequency domain impression of a masked area has a much greater spread than simple low frequency undulations in intensity (due to the step edge bordering the masked area). If a background image is employed then the transform of the masked area, as well as the low frequency intensity fluctuations, are eliminated in the subtraction of spectra. In the windowing case it is possible that the masked area impinges on the side lobe so that the side lobe is not correctly isolated.

2.6 Phase Unwrapping Algorithms

The following section describes a number of algorithms which have been developed for phase unwrapping.

The reader is asked to bear in mind that most of the algorithms rely on the successful detection of fringe edges. Considering this fundamental point, it is surprising how little attention has been paid to the subject. Edge detection will be covered in Chapter Four.

2.6.1 The Phase Fringe Counting/Scanning Approach To Phase Unwrapping

The procedural steps followed under a Fringe Counting system are described below:

- i) The digitised interferograms are filtered to eliminate noise. As for the fringe tracking method, this process may be performed by spatial filtering (see Chapter Four) or an FFT low pass filter [39]. A wrapped phase map is then computed by either the Quasi-Heterodyne or FFT techniques.

The field then contains a relatively clear definition of fringe boundaries. The phase is linearly related to the measurement parameter other than at these edge discontinuities.

- ii) The fringe edges may be found via an edge detection system which seeks a phase change of a specified threshold level. A binary image recording these points is then produced.
- iii) The field is retraced on a scan line by scan line basis, a count is kept of the fringe edges passed using the binary image found above. This count is incremented or decremented dependent on the direction of rollover of the fringe edge. The count (multiplied by 2π) is added to the phase values of successive pixels [11].
- iv) The horizontal scans are then arranged relative to one another by using a single vertical scan, or vice versa. Various strategies may be applied to select a good candidate for the arranging scan.

- v) At the conclusion of this process an unwrapped map is obtained.
By scaling the range of the unwrapped phase, an estimate of the measurement parameter is produced.

The algorithm relies on the quality of the fringe edges. However, there can be no guarantee that, even after filtration, noise in the fringe edges will disappear. The behaviour of the edge detection technique is an important factor. For example the fringe edge must always be distinguished by at least the threshold level of the detection procedure. Any failure is carried across the field and may disrupt many points of a particular scan in the map.

The scanning algorithm, as was mentioned earlier, has little hope of resolving a discontinuous fringe field as, in the case where holes exist, simple fringe counting would far from represent the measurement parameter. Many errors would be propagated across the field.

2.6.2 Cellular-automata Method for Phase Unwrapping

In the simplest case, a cellular automaton consists of a line of sites, each site having a value zero or one. The sequence of site values is the 'configuration' of the cellular automaton. The cellular automaton evolves in discrete time steps. At each time step, the value of the site is updated according to a definite rule. The rule specifies the new value of a particular site in terms of its own old value, and the old values of sites in some neighbourhood around it. The neighbourhood is typically taken to include sites up to some small finite range from a particular site. In general, the sites in a cellular automaton may take on any finite set of possible values, rather than simply zero and one. In addition, the sites may be arranged on a two or higher dimensional lattice, rather than on a line. As a further generalisation, one may allow the value of a particular site to depend not only on values at the previous time step, but also on values from preceding time steps [47].

Cellular automata have five fundamental defining characteristics:

- i) They consist of a discrete lattice of sites.
- ii) They evolve in discrete time steps.

- iii) Each site takes on a finite set of possible values.
- iv) The value of each site evolves according to the same deterministic rules.
- v) The rules for the evolution of a site depend only on a local neighbourhood of sites around it.

Ghiglia describes an interesting automaton which unwraps phase [13]. However, it should be noted at the outset that it is not the action of the automaton logic itself which detects or flags errors in the phase map. The automaton performs the task over a large number of iterations. Considering a two dimensional field, the whole field area progresses successively towards the unwrapped solution over each iteration.

A field is unwrapped by the action of many operations, which to begin with have a small spatial extent. It is assumed that the field fundamentally represents a continuous function, but with added pointlike inconsistencies.

The approach works from the pixel level and gradually gathers more and more of the surrounding areas together, until the whole field is solved.

2.6.2.1 Detecting Possible Inconsistencies

A simple means of flagging possible inconsistent regions is implemented by checking all 2×2 pixel areas in the field according to the procedure shown in Figure 2.16. The phase is unwrapped along the closed path indicated. If the sum of the wrapped-phase differences along the path equals zero, then all four points are said to be consistent; otherwise all four points are flagged as inconsistent.

This is done for all 2×2 regions until the entire field is covered. These flagged inconsistent regions may be superimposed upon the wrapped phase map. Depending on the type of inconsistency, the 2×2 pixel consistency check may not find all inconsistent paths, but gives visual evidence where problems can be expected.

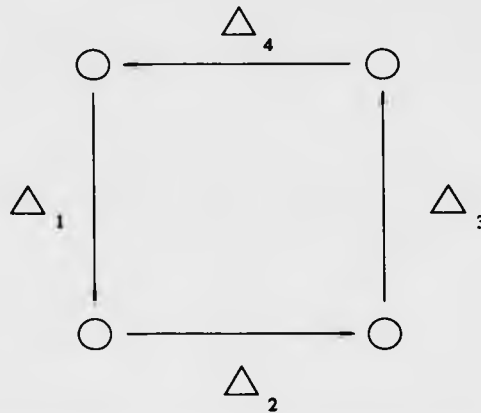


Figure 2.16: 2 by 2 Pixel, Path Consistency Check

2.6.2.2 Two-Dimensional Algorithm Description

Any point flagged as inconsistent by the 2×2 path checker (or by manual intervention if required) is not included in any automaton decision-making process. Only the unflagged points are used in the local neighbourhood. The complete 2-D algorithm for the automaton is as follows:

- i) Each site looks at all unflagged neighbours within a distance of one unit. Only 4-connected (i.e. orthogonal) neighbours are allowed, not diagonal neighbours. The phase differences between the site of interest and each neighbour are computed (i.e. $\phi_{site} - \phi_{neighbour}$).
- ii) The strength of each neighbour's vote is defined to be equal to the integer number of 2π shifts necessary to wrap the respective phase differences. The strength of votes is accumulated.
- iii) The site is changed in value by 2π in a direction appropriate to the accumulated strength-of-vote (i.e., if there is a stronger 'up' vote than 'down' vote, the site increases by $+2\pi$).
- iv) If none of the neighbours differ by more than π from the current site, no change is made to the site value. If the accumulated

strength-of-vote is identically zero, the site is changed by $+2\pi$ (i.e. tied strength-of-votes are arbitrarily broken in the positive direction).

- v) Repeat steps i) through iv) until the period-two oscillatory state is reached, then average two oscillatory states [13]. If the phase is not unwrapped, go to first step; otherwise terminate.

2.6.2.3 Cellular-automata Conclusion

Cellular-automata methods can unwrap phase in one and two dimensions, beginning with an array of principal values. The cellular-automata method has a natural parallelism and path independence.

It is pointed out, however, that the ability of the technique to deal with noise points, relies upon proper masking by the 2×2 noise detection procedure. This is not part of the cellular automaton logic. Any bad areas of data in the interferogram would have to be masked, via some algorithm, or manually for the cellular automaton to operate correctly.

In contrast, the weighting strategy of the MST algorithm does not require the flagging of point noise by a 2×2 pixel consistency check. The MST algorithm assumes that the larger the phase change between adjacent pixels, the more likely an error becomes.

It is stated that aliasing-induced or natural phase dislocations can be accommodated by region partitioning guided by *a priori* knowledge or by almost arbitrary partitions designed to prevent unwrapping across inconsistent boundaries. The MST algorithm will show that such partitioning may be achieved to a great extent automatically.

As the algorithm works close to the pixel level, automatic partitioning of large areas is not considered. In the absence of any inserted partitions the algorithm produces a series of self-similar patterns, where some discontinuities are removed and others recreated in a never-ending, extremely long period cycle.

A second disadvantage of the algorithm is that it may require a very large number of iterations to produce a result. In addition each iteration is inherently parallel in nature and so on a sequential machine takes a substantial

time to complete for all cells. It is stated that parallel machines may alleviate this problem. The use of a larger cell size would also increase the speed of the algorithm. This requires more rules. Since Ghiglia first published his work, there has been some interest in the method.

Buckberry [48], has demonstrated a cellular-automata phase unwrapping system which employs the power of a frame store with an associated arithmetic processor. This processor permits rapid execution of low level pixel operations, and has proved highly suitable for the technique. The approach is employed in an ESPI system, employing the phase stepping method.

Spik [49] has investigated the method and implemented it on an array processor. Spik makes some comments on the operation of the algorithm. The algorithm begins modifying the field from the edges of phase fringes. It shifts these edges by one cell in each pass through the field. This process is called 'local iteration'. Eventually, a period-two oscillatory state is reached, which can be detected by comparing the latest iteration with the previous one. To continue phase unwrapping, a global iteration has to be applied, during which the mean value for each point is calculated by taking values from the latest picture and the previous one. The field, after a global iteration, appears similar to that before the global iteration except that phase fringe edges are shifted to different positions and often lie between the fringe edges of the initial condition. The fringe nearest to the bounds of the frame is removed, the global iteration removes one fringe from the wrapped phase. The number of global iterations necessary to completely unwrap the phase is greater than or equal to the number of phase fringes. Large scale discontinuities produce changes during local iterations and finally unwanted fringes. If no correction routine is applied the phase is not properly unwrapped. In these cases algorithms never reach a stationary state and manual intervention is necessary.

Spik further states that the number of local iterations necessary to reach the period-two oscillatory state depends on the number of fringes and their locations in the field. For many phase fringes the number of local iterations is small because the cellular distributions which spread from fringe edges meet similar cellular structures from neighbouring fringes. If fringes are close together then a few local iterations fill the space between them and it quickly

tends to the period-two oscillatory state. After several global iterations, when few fringes remain in the field, the number of local iterations required to complete all global iterations depends on the position of the outer fringe in relation to the field boundary. The cellular pattern also spreads from the outer fringe edge to the boundary. In this particular case the number of local iterations necessary before the global iteration may equal the image resolution [49].

2.6.2.4 Modification to Original Cellular Automata Algorithm

Spik describes a modification to the original algorithm to aid with the detection of inconsistencies. The original algorithm, due to Ghiglia, has no mechanism to remove or mask large-scale path inconsistencies. The routine to mask path inconsistencies consists of a test of every 2×2 group of pixels to ensure that the sum of the wrapped phase differences in a square path around all four pixels is zero. If an inconsistency exists, the sum will be non-zero and all four are masked out. This procedure, in the original algorithm, is only applied once, before the phase unwrapping process. This is inadequate because the routine masks single point path inconsistencies rather than larger dislocations. If such a larger phase dislocation appears in the field, only the ends of the discontinuity line are masked, not the whole path. The discontinuity masking algorithm can be improved by applying the same routine after each global iteration, thus allowing the masked points to propagate along the whole length of an inconsistency or fringe break. Spik states that the improvement is not directly applicable to the original algorithm. The original algorithm has therefore been modified.

The modification involves consideration of the 8 neighbours to each point, instead of only 4. The result is to enlarge the simple cell by a factor of two, compared to the original one. The speed of the algorithm is improved by almost a factor of two. After each global iteration phase fringes are obtained without striation lines. This permits the phase inconsistency masking routine to be applied after each global iteration [49].

There is a shortcoming in the algorithm which should be pointed out. Even applying the masking routine after each global iteration, discontinuities

which cause a large fringe break are not catered for, also the direction in which a discontinuity should be propagated is not determined by the masking routine.

The spatial direction in which phase unwrapping proceeds, on an iteration by iteration basis, with the cellular automaton is not affected by the quality of the data in the field above the size of the cell or of the masking test. That is all areas are unwrapped simultaneously and synchronously. The routine does not try to avoid areas of inconsistency, it simply runs into them, avoiding those that may be detected by pixel comparisons over a number of iterations. Actually Spik mentions this limitation, 'for long discontinuities phase map partitioning may occur'.

2.6.3 Berlin Development of Minimum Spanning Tree Method

Since beginning this literature survey it has come to my attention that a group in Germany have also been working with Minimum Spanning Trees for phase unwrapping [14].

Their first paper on the subject, as far as the author is aware, was first presented in April 1989 [14]. The author's early work on minimum spanning trees for phase unwrapping was first presented in August 1989 [50]. A version of this paper incorporating the tiling method, but without the minimum spanning tree connection strategy was presented in March 1989. Two journal papers by the author have appeared covering stages in the development of the MSTT phase unwrapping technique these are given as references [51] and [52].

The manner in which Ettemeyer, the author of the Berlin paper, employed the trees differs from the way the author first employed them. Ettemeyer, again, chose to address the pixel level phase unwrapping problem. The author first used tiles, or blocks of pixels, to solve around larger discontinuities. The Berlin paper was published in German. The relevant section has been translated, and this appears below. It is similar to the pixel level phase unwrapping algorithm (discussed in Chapter Three).

2.6.3.1 The Berlin Pixel Level Minimum Spanning Tree Algorithm

The basic principle of the phase unwrapping method is that before the procedure of phase unwrapping, there is a path to be found through the phase image where errors can be avoided with the most probability. Parts with very bad information should be unwrapped at the very end of the evaluation so possible mistakes are localised and not dragged along through the whole picture.

To do this, the computer first of all compares the complete neighbourhoods in the modulo- 2π picture and sorts them with the criteria of 'size of phase difference between neighbouring pixels'.

A graph of edges (neighbourhoods) is built up so every point can be compared to their upper right neighbour. Afterwards a table is constructed where edges are arranged so that the edges with the smallest values (the difference of the phase between neighbouring points is as small as possible) are at the top and points with increasing values are at the bottom. Of importance here, is consideration of the phase being arranged in a unified circle so the values 0 and 360 degrees are identical.

With the help of this table a minimum spanning tree is constructed. That is all neighbouring points with small edges are connected ascending to the greater edge values. This means, that points with the greatest neighbourhoods are evaluated at the very end. However, because those are exactly the points with the highest probability of containing mistakes, these are limited locally to small parts of the phase map.

This is, for example, especially recognisable in areas where the sampling theorem is violated, e.g. if the density of edges is too great, or there are cracks in a part of the reconstruction (interference fringe jumps), or shadows (missing information). [14]

2.6.3.2 Conclusion to the Berlin Pixel Level Minimum Spanning Tree Algorithm

The method described above has the same underlying principle to my own pixel level unwrapping algorithm. The weights are computed differently,

however, that is here the weights are computed from the difference in the average phase of adjacent pixel pairs, rather than the difference of pixels, only. From my investigations, the latter weighting strategy is more sensitive to noise. This may be simply explained.

The effect of an *averaging* filter, placed over a spike noise point, is to spread the spike so that it raises the values of surrounding pixels. Using the average in the graph weighting strategy serves to increase the weight of graph edges, which connect to pixels surrounding the noise point. These are embedded in the unwrapping path *before* the noise spike is actually reached, thereby causing the algorithm to delay further its approach to the noise spike.

The German algorithm is, without the benefit of the hierarchical approach, sensitive to discontinuities larger than the pixel. The algorithm attempts to solve the entire field in one swoop. A major problem is the time complexity of the minimum spanning tree algorithm. The best sequential algorithm has a time complexity of $O(n^2)$, where n is the number of pixels in the image (the same order as a bubble sort). This means that the execution time of the algorithm increases as the square of the number of pixels considered. If the whole image is considered, as Ettemeyer implies, then the phase unwrapping algorithm will perform very slowly. Using the tiling approach, the number of pixels considered at one time is restricted to the area of the tile, and so n is reduced. That is the complexity of the algorithm becomes $O(\lceil \frac{n}{m} \rceil m^2)$ where m is the number of pixels in a tile, and n the pixels in the image (this figure does not include the overlap pixels).

2.6.4 Noise-immune Cut Methods of Phase Unwrapping

The phase unwrapping method described by Goldstein [15] is an enhancement of the fringe scanning algorithm. The basis of the algorithm is to place cuts, on a scan line by scan line basis, between points of phase discontinuity in order to minimise the length of the propagated discontinuity. The size of the discontinuity is termed the cut length.

The point discontinuity masking procedure, described with regard to the

Cellular-Automata method, is also used in this algorithm. However, the masked point is now known as a residue, and is signed. That is it may be either positive by one cycle, negative or zero. This is illustrated by Figure 2.17 (the values shown are fractions of a fringe, not phase values). The sign of the residue is determined by the difference between the start and end phase pixels, after a clockwise step through the 4 pixels comprising the residue. The algorithm is described below.

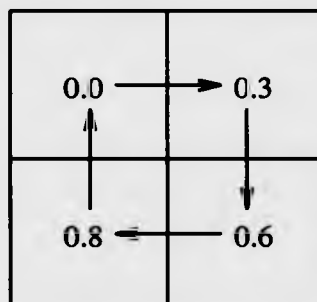


Figure 2.17: A Positive Residual

The interferogram is scanned until a residue is found. A box of size 3 pixels is placed around the residue and searched for another residue. If another residue is found, a cut is placed between the pair detected. If the new residue is of opposite sign to the first, then the cut is designated as "uncharged" and the scan is continued in search of another residue. If, however, the sign of the residue is the same as the original, then the box is moved to the new residue and the search is continued until either an opposite residue is located, the resulting total cut then being uncharged, or new residues can be found within the boxes. In the latter case, the size of the box is increased by 2 and the algorithm repeats from the then current starting residue.

The processing stage above detects pixels of phase discontinuity and masks them out. The phase unwrapping path required to circumnavigate the discontinuities is still to be computed. The next step in the process is a little vague. Goldstein implies that whilst the masking process is underway, phase is unwrapped scan line by scan line as far as the first residue on each scan line. Thus, a portion of the field would be unwrapped upon completion

of the masking process, although these partial scan line solutions would not be related to one another. It seems that a kind of flood fill of the remaining pixels is applied, taking the points already unwrapped as the initial wave front. This process would have to take account of the remaining phase roll over points.

Goldstein states that a 'layover' area in their satellite images was typically characterized by a preponderance of residues of similar sign, in a line, in one half of the layover region, with a corresponding set of opposite sign in the other half. It is stated that the cutting algorithm was successful in isolating such areas.

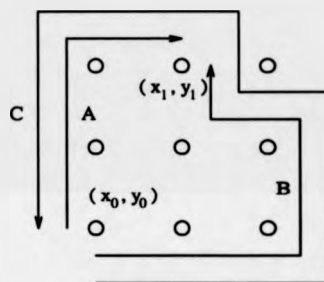


Figure 2.18: Two Alternative Pixel Paths for Unwrapping the Phase at Data Point x_1, y_1

Huntley [16] has also described an algorithm which relies on placing cuts within the phase map. The algorithm is aimed at solving consistent phase maps with noise spikes, although the algorithm would also isolate adjacent residues, in a similar manner to the method described by Goldstein above.

The basis of Huntley's algorithm is the requirement that, given the phase at pixel (x_0, y_0) , the phase at any other point (x_1, y_1) in the image should be defined uniquely, independent of the path by which the phases are unwrapped. This is again achieved by placing cuts in the phase map, which act as barriers to unwrapping.

Consider the two pixel paths A and B in Figure 2.18. The sequences of phase values along the two paths are denoted $\Phi_A(i)$ ($i = 0, 1, \dots, N_A$) and $\Phi_B(j)$ ($j = 0, 1, \dots, N_B$), respectively ($N_A = 3$ and $N_B = 5$ in Figure 2.18). Unwrapping along A is achieved by calculating the number of 2π

discontinuities, $d_A(i)$ ($i = 1, 2, \dots, N_A$), between adjacent pixels:

$$d_A(i) = [(\Phi_A(i) - \Phi_A(i-1))/2\pi] \quad (2.51)$$

where [...] denotes rounding to the nearest integer. The value $2\pi d_A(i)$ is then subtracted from the phase values along the rest of the path (i.e. from $\Phi_A(i')$, $i' = i, i+1, \dots, N_A$). The sequence $d_B(j)$ required to unwrap Φ_B is defined in a similar way. Uniqueness of the unwrapped phase at (x_1, y_1) requires the total number of discontinuities along the two paths be equal; i.e. that the parameter S , defined below, is equal to zero,

$$S = \sum_{j=1}^{N_B} d_B(j) - \sum_{i=1}^{N_A} d_A(i) \quad (2.52)$$

If path A is reversed, the $d_A(i)$ all change sign, so that S is just the total number of 2π discontinuities around the counter clockwise loop C . The problem, therefore, is to construct the cut lines such that any permissible closed loop (i.e. one which does not cross a cut) has $S = 0$.

To proceed systematically, a closed loop around each of the smallest possible units of the phase map: a square of 4 pixels, is considered. The distribution of s (the discontinuity source map) is calculated from $\Phi(x, y)$ as follows:

$$\begin{aligned} s(x, y) = & [(\Phi(x+1, y) - \Phi(x, y))/2\pi] + \\ & [(\Phi(x+1, y+1) - \Phi(x+1, y))/2\pi] + \\ & [(\Phi(x, y+1) - \Phi(x+1, y+1))/2\pi] + \\ & [(\Phi(x, y) - \Phi(x, y+1))/2\pi] \end{aligned} \quad (2.53)$$

The value of S for large loops can be easily obtained from $s(x, y)$. For example, path C in Figure 2.18 has $S = s(x_0, y_0) + s(x_0+1, y_0) + s(x_0, y_0+1)$ because the contributions from the internal paths cancel. In general, S can be calculated for any closed loop as

$$S = \sum_{x,y} s(x, y) \quad (2.54)$$

where the sum is over all pixels enclosed by the loop [16].

In implementing the algorithm, Huntley uses two arrays of flags, $H(x, y)$

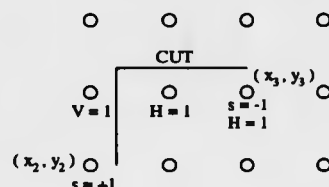


Figure 2.19: Example Cut Made Between Two Discontinuity Sources $s = +1$ and $s = -1$

and $V(x, y)$. These are initially set to zero. To indicate a cut between a pair of points, for example (x_2, y_2) and (x_3, y_3) , the vertical cut array $V(x_2, y)$ is set to 1 for all points with $x = x_2$ and y between either $y_2 + 1$ and y_3 , if $y_2 < y_3$ or $y_3 + 1$ and y_2 , if $y_3 < y_2$. A similar procedure is performed for the horizontal cuts, setting $H(x, y_3)$ to 1 for all points with $y = y_3$ and x between either $x_2 + 1$ and x_3 , if $x_2 < x_3$ or $x_3 + 1$ and x_2 , if $x_3 < x_2$. A simple example is given in Figure 2.19.

Phase unwrapping may then be carried out from any point, as is necessary for path independence. For example, suppose the phase at point (x, y) has been unwrapped but that at $(x+1, y)$ has not. A valid path is first established between the two points. Normally this would have a single link: the number of discontinuities would be calculated as $d = [(\Phi(x+1, y) - \Phi(x, y))/2\pi]$, and $2\pi d$ would be subtracted from $\Phi(x+1, y)$. However, if $V(x, y) = 1$, indicating a vertical cut between the points, the search direction is rotated 90 degrees counter clockwise, to the point $(x, y+1)$. This is the next valid point in the path, provided $H(x, y) = 0$. Each successive link is established by rotating the search direction through 90 degrees clockwise compared with the previous link. In this way, cuts are circumnavigated in a clockwise direction. Any other path (e.g., counter clockwise circumnavigation) is of course also valid. The number of 2π phase discontinuities between the successive elements along the path $d(i)$ is calculated according to Eqn. 2.51, and $2\pi \sum d(i)$ is subtracted from $\Phi(x+1, y)$.

The ability to deal with regions of an image that contain no fringe information is one of the main advantages of the technique.

2.6.4.1 Cut Method Conclusion

The cut algorithms rely essentially on one test of consistency, typified by Goldstein's residue, which is uniquely aimed at detecting spike noise. The area over which the consistency test operates is very small, 2×2 pixels. Huntley's algorithm amalgamates the results of many such tests, but the general problem of 'natural or aliasing induced path inconsistencies' remains unaddressed.

Natural or aliasing induced discontinuities do not show themselves over the small test area considered, because the image areas containing such discontinuities do not always contain non zero residues. Several examples of these types of discontinuity will be seen in Chapters 3 and 5. The cut algorithms deal with inconsistencies between discrete points. There is no strategy to consider large scale regional effects.

2.6.5 Phase Unwrapping by Regions

Gierloff [53] has proposed a phase unwrapping algorithm which differs significantly from the Cellular-Automaton, Minimum Spanning Tree, and Cut methods just described. This method attempts to segment the fringe field into areas of consistency, and then to relate these areas to one another. This method recognises the problem posed by large scale discontinuities.

A first attempt to unwrap the fringe field is made via a procedure similar to Fringe Counting, or at least Gierloff implies this. The algorithm then operates by dividing the fringe field into regions, see Figure 2.20. Regions are decided by determining whether points lie within a tolerance of adjacent points already included. A point is considered to be part of a region if a given percentage of adjacent points are within this tolerance band. A point may have up to 8 neighbours. Typical values for percentage agreement have been from 40 to 65 percent and typical tolerances from 0.5 to 1.5 radians (one fringe = 2π radians).

Once all points have been assigned to a region, the edges of the regions are compared to determine if there is a discontinuity between them. The regions may have logical inconsistencies. For example, some points along region edges may be within an acceptable range of each other and yet others

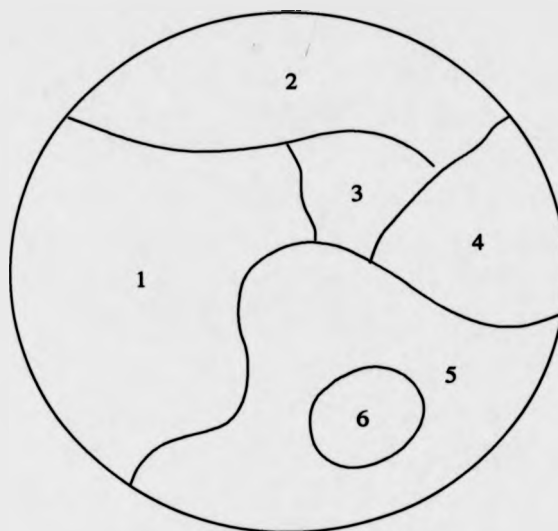


Figure 2.20: Field Divided into Regions

not. Some neighbouring regions may suggest that a given region does not need to be phase shifted whilst others might suggest that it should.

All edges between adjacent regions are traced. The edge points are compared to determine whether a shift should be made, up by 2π , down by 2π or whether no shift. Each point carries a vote. The number giving a specific answer must be greater than a defined fraction of the total number of points in the edge.

Once the relationships between regions have been defined, regions that have been identified as having no phase ambiguities are combined into a single larger region. These larger regions are then compared to determine necessary phase shifts.

It is possible to have logical inconsistencies when adjusting regions relative to other regions. The determination of how to shift regions is made by weighting the answer of the comparison by the number of points along the common edge [53].

2.6.5.1 Region Method Conclusion

In concluding Gierloff states

- i) The algorithm did not give perfect performance, although it outperformed several algorithms based on Fringe counting techniques.
- ii) The reasons underlying the algorithm's failures were not understood.
- iii) There seemed to be no fundamental reason why the technique could not produce results that were close to the globally optimal.
- iv) Resolution of the remaining problems would require an in-depth examination of phase shifting errors.

Problems with the approach are

- i) Path independence is not guaranteed. That is, for example, the algorithm could produce substantially different solutions depending upon the particular region from which phase unwrapping begins. For example starting from a region with aliasing induced inconsistencies (which would not show within the region) would mean that adjoining regions would be mismatched. In the MSTT algorithm, the MST guarantees that connection errors are minimised when regions are assembled.
- ii) The weighting criterion for connecting regions together is flawed. As the interferogram is two dimensional it is not sufficient to compare merely the length of an edge. A straight edge may cover a substantially greater *area* than a circular one of the same length, and so give incompatible indications of confidence. More mathematically, the metric for comparing confidence is unsound.

2.6.6 Phase Unwrapping Using a Priori Knowledge about the Band Limits of a Function

This approach to phase unwrapping uses information not considered in the algorithms described above, that is, knowledge of the frequency band limits

of a wrapped phase map.

Green [54] describes an algorithm based on this information and demonstrates some one dimensional examples where it is successful and a fringe scanning algorithm fails. It is also shown that the band limited algorithm fails in some cases where the fringe scanning algorithm is successful. It is proposed that a robust and practical algorithm could be constructed as a hybrid incorporating both fringe scanning and the band limited approach.

When a phase function is wrapped, it is made modulo 2π . If a phase function which is frequency band limited is wrapped, then the resulting function has a spectrum which extends beyond the bandlimits [54]. The bandlimited phase function ϕ can be written in terms of the wrapped phase ϕ_w

$$\phi(x) = \phi_w(x) + g_n(x) \quad (2.55)$$

where $g_n(x)$ is a function with a number of steps n . The Fourier transform of equation 2.55 gives

$$\Phi(f) = \Phi_w(f) + G_n(f) \quad (2.56)$$

From equation 2.56 the Fourier transform of the wrapped phase is given by

$$\Phi_w(f) = \Phi(f) - G_n(f) \quad (2.57)$$

The spectrum of the unwrapped phase is limited between the frequencies $-f_1$ and $+f_1$. It is the step function which extends the frequency band beyond these limits when the phase function is wrapped.

Given that

$$s(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$

consider the effect of adding another step function at an arbitrary position to the original step function $g_n(x)$

$$g_{n+1}(x) = g_n(x) + 2\pi a_{n+1} s(x - x_{n+1}) \quad (2.58)$$

where a_{n+1} can either be $+1$ or -1 . This has the Fourier transform

$G_{n+1}(f)$ where

$$G_{n+1}(f) = G_n(f) + (2\pi\delta(f) + \frac{1}{if})a_{n+1}e^{ifx_{n+1}} \quad (2.59)$$

or

$$G_{n+1}(f) = (2\pi\delta(f) + \frac{1}{if})C_{n+1}(f) \quad (2.60)$$

for

$$C_{n+1}(f) = \sum_{k=1}^{n+1} a_k e^{i2\pi x_k f} \quad (2.61)$$

As only the contribution of the step function is present beyond the band limits, it is expected that eliminating a phase discontinuity would reduce the modulus of the out-of-band spectrum for all f relative to the spectrum of the wrapped phase. In this case

$$|C_{n+1}(f)| < |C_n(f)| \quad (2.62)$$

The inequality in equation 2.62 is not always true, as the removal of a step may result in an increase in the modulus of the resulting complex phasor sum. However, an increase in the modulus becomes less likely as the resultant modulus decreases, that is, as the number of steps decreases. Since ultimately the result of removing all steps is zero out-of-band information then it is probable that equation 2.62 will be true for all f .

A measure of the out-of-band power in $\phi_w(x)$ is

$$\sum_{|f|>f_1} |G_n(f)| \quad (2.63)$$

which is effectively an average of $|G_n(f)|$ over all frequencies outside the band limits. This sum is easily calculated from the data. The removal of a phase discontinuity implies that the inequality

$$\sum_{|f|>f_1} |G_{n+1}(f)| < \sum_{|f|>f_1} |G_n(f)| \quad (2.64)$$

is true.

2.6.6.1 The Band Limited Phase Unwrapping Algorithm

Given a single dimensional array of wrapped phase values $\phi_w(x)$ as data, all possible step functions defined by the combination (a_{n+1}, x_{n+1}) are added to obtain all possible functions $g_{n+1}(x)$. These are Fourier transformed and the particular combination which minimises the sum

$$\sum_{|f| > f_1} |G_{n+1}(f)| \quad (2.65)$$

is kept and assumed to define a step function which unwraps a phase discontinuity. The corresponding 2π phase discontinuity is corrected for and the procedure is repeated to correct for further 2π phase discontinuities until the addition of further steps no longer reduces the sum of the out-of-band moduli.

2.6.6.2 Band Limited Phase Unwrapping Algorithm Conclusion

For wrapped functions corrupted by additive noise, the algorithm is able to make correct decisions as to whether a feature is a 2π phase discontinuity or not, where a fringe scanning algorithm will fail. This is because fringe scanning phase unwrapping algorithms operate on local neighbourhoods of the wrapped phase and local smoothness is based upon the values of differentials in a particular neighbourhood. This algorithm operates on a global basis and global smoothness is based upon the amount of frequency information outside the band limits.

2.7 Conclusion

This chapter has given some background on interferometric methods, with particular reference to phase unwrapping.

Some concepts from the various phase unwrapping algorithms discussed above are taken up in the MSTT algorithm, which is discussed in depth in the next chapter. These are the ideas of pixel level phase unwrapping, encompassed by the Cellular-Automata and Cut Methods, and the concept of regional phase unwrapping represented by the Region Method.

The MSTT algorithm has two hierarchical levels, a regional level and a pixel level. Each level has advantages, at the pixel level a close inspection of pixel phase values permits circumvention of spike noise, the regional level permits the effect of large scale discontinuities to be seen and avoided.

Bibliography

- [1] G. T. Reid, "Image Processing Techniques for Fringe Pattern Analysis", Proceedings of the 1. International Workshop on Automatic Processing of Fringe Patterns, Berlin, pp. 13-20, 1989.
- [2] R. Thalman and R. Dandliker, "High Resolution Video Processing For Holographic Interferometry Applied To Contouring And Measuring Deformations," SPIE ECOOSA, vol. 492, Amsterdam, 1984.
- [3] B. Breuckmann, W. Thieme, "Computer-aided Analysis of Holographic Interferograms Using the Phase-shift Method", Applied Optics, Vol. 24, No. 14, pp. 2145-2149, 15 July 1985.
- [4] R. Dandliker, R. Thalmann, "Heterodyne and Quasi-heterodyne Holographic Interferometry", Optical Engineering, Vol 24., No. 5, pp. 824-831, September/October 1985.
- [5] G. T. Reid, "Automatic Fringe Pattern Analysis: A Review," Optics and Lasers in Engineering, vol. 7, pp. 37-68, 1986/7.
- [6] M. Takeda, H. Ina, and S. Kobayashi, "Fourier-transform Method Of Fringe-Pattern Analysis For Computer-based Topography And Interferometry," Journal Of The Optical Society Of America, vol. 72, pp. 156-160, 1981.
- [7] M. Kujawinska, J. Wojciak, "High Accuracy Fourier Transform Fringe Pattern Analysis", Applied Optics Digest, 17th-20th September, pp. 257-258, 1990.
- [8] B. L. Button, J. Cutts, B. N. Dobbins, C. J. Moxon, and C. Wykes, "The Identification Of Fringe Positions In Speckle Patterns," Optics

And Laser Technology, vol. 17, pp. 189-192, 1985.

- [9] W. R. J. Funnell, "Image Processing Applied To The Interactive Analysis Of Interferometric Fringes," *Applied Optics*, vol. 20, pp. 3245-3249, 1981.
- [10] S. Nakadate, T. Yatagai, and H. Saito, "Computer Aided Speckle Pattern Interferometry," *Applied Optics*, vol. 22, pp. 237-243, 1983.
- [11] S. Nakadate and H. Saito, "Fringe Scanning Speckle-pattern Interferometry," *Applied Optics*, vol. 24(14), pp. 2172-2180, 1985.
- [12] D. W. Robinson and D. C. Williams, "Digital Phase Step- ping Speckle Interferometry," *Optics Communications*, vol. 57(1), pp. 26-30, 1986.
- [13] D. C. Ghiglia, G. A. Mastin, and L. A. Romero, "Cellular-Automata Method For Phase Unwrapping," *J. Opt. Soc. Am.*, vol. 4A, pp. 267-280, 1987.
- [14] A. Ettemeyer, U. Neupert, H. Rottenkolber, C. Winter, "Schnelle Und Robuste Bildanalyse Von Streifenmustern - Ein Wichtiger Schritt Der Automation Von Holografischen Prozessen", *Proceedings of the 1. International Workshop on Automatic Processing of Fringe Patterns*, Berlin, pp. 23-31, 1989.
- [15] R. M. Goldstein, H. A. Zebker, C. L. Werner, "Satellite Radar Interferometry: Two-dimensional Phase Unwrapping", *Radio Science*, Volume 23, Number 4, Pages 713-720, July-August, 1988.
- [16] J. M. Huntley, "Noise-immune Phase Unwrapping Algorithm," *Applied Optics*, vol. 28(15), pp. 3268-3270, 1989.
- [17] E. Hecht, *Optics*, Addison-Wesley, 1987. ISBN 0 201 11611 1
- [18] W. E. Williams, *Applications Of Interferometry*, Methuen & Co. Ltd., London, 1961.
- [19] R. Crane, "Interference Phase Measurement," *Applied Optics*, vol. 8, pp. 538-542, 1969.

- [20] P. Hariharan, Optical Holography, Cambridge University Press, 1984.
ISBN 0 521 24348 3
- [21] P. S. Theocaris, Moire Fringes In Strain Analysis, Pergamon Press, Oxford, 1969.
- [22] L. C. Squire, P. J. Bryanston-Cross, X. Liu, "An Interferometric Study of the Shock Interaction at a Compression Corner", Aeronautical Journal, pp. 143-151, May, 1991.
- [23] R. J. Parker, "Use of Holographic Interferometry for Turbomachinery Fan Evaluation During Rotating Tests", Journal of Turbomachinery, Vol. 110, No. 3, pp. 393-400, Jul 1988.
- [24] M. P. Escudier, J. A. Bornstein, "Experimental Investigation of Transonic Flow in 2-Dimensional Turbine Cascades", Brown Boveri, Forschungszentrum, CH-5405, Baden, Germany, KLR 83-183 B, March 1984.
- [25] D. Wiedow, "Application of Holographic Interferometry in Car Development", Papers Presented at the SAE Passenger Car Meeting and Exposition, Dearborn, MI, USA, Oct 19-22, SAE Technical Paper Series, Published by SAE, Warrendale, PA, USA Pap 871945, 10p, 1987.
- [26] T. Manderscheid, "Adjustment of Sonotrodes by Holographic Interferometry", SPIE Vol. 599, Optics in Engineering Measurement, pp. 40-45, 1985.
- [27] Colloquium on Optical Techniques for NDT, IEE Colloquium (Digest) n 1986/12, Published by IEE, London, Jan 1986.
- [28] Ovrin, Benji, "Holographic Interferometry", CRC Critical Reviews in Biomedical Engineering, Vol. 16, No. 4, pp. 269-322, 1989.
- [29] S. N. Panov, "Use of Holography in the Study of Metal Cutting Machine Tools", Soviet Engineering Research, Vol. 7, No. 8, pp. 39-43, Aug 1987.
- [30] N. Rubayi, J. Yeakle, M. A. Wright, "Holographic Interferometry as a Non-destructive Method in Detecting Flaws in Composite Plates",

Advanced Composites: The Latest Developments, Proceedings of the Second Conference, Dearborn, MI, USA, Nov 18-20, Published by ASM Int, Metals Park, OH, USA, pp. 269-270, 1986.

- [31] M. A. De Smet, "Holographic Non-destructive Testing for Composite Materials Used in Aerospace", SPIE Vol. 599, Optics in Engineering Measurement, pp. 46-52, 1985.
- [32] R. J. Pryputniewicz, "Holographic and Finite Element Studies of Vibrating Beams", SPIE Vol. 599, Optics in Engineering Measurement, pp. 54-62, 1985.
- [33] P. Jung, "Contribution of Holographic Interferometry to the Numerical Simulation of Concrete Shrinkage", SPIE Vol. 599, Optics in Engineering Measurement, pp. 32-39, 1985.
- [34] H. Steinbichler, J. Engelsberger, W. Sixt, J. Sun, Th. Franz, "Application of Computer-Aided Evaluation for Holography and Similar Techniques", Optics and Lasers in Engineering 13, pp. 39-50, 1990.
- [35] D. Paoletti, G. Schirripa, "Sandwich Hologram: A Practical Tool for Stress Analysis in Paintings on Canvas", SPIE Vol. 599, Optics in Engineering Measurement, pp. 105-110, 1985.
- [36] A. C. Gillies, "Using Knowledge to Improve the Scope and Efficiency of Image Processing in Fringe Analysis", SPIE Vol. 1095 Applications of Artificial Intelligence VII, pp. 742-749, 1989.
- [37] T. Yatagai, M. Idesawa, Y. Yamaashi, and M. Suzuki, "Interactive Fringe Analysis System: Applications To Moire Contourgram And Interferogram," Optical Engineering, vol. 21, pp. 901-906, 1982.
- [38] T. M. Kreis, "Quantitative Evaluation Of Interference Patterns," SPIE Industrial Optoelectronic Measurement Systems Using Coherent Light, vol. 863, 1987.
- [39] D. A. Chambless and J. A. Broadway, "Digital Filtering Of Speckle Photography Data," Experimental Mechanics, vol. 19, pp. 286-289, 1979.

- [40] J. B. Schemm and C. M. Vest, "Fringe Pattern Recognition And Interpolation Using Non-Linear Regression Analysis," *Applied Optics*, vol. 22, pp. 2850-2853, 1983.
- [41] T. Yatagai, S. Nakadate, M. Idesawa, and H. Saito, "Automatic Fringe Analysis Using Digital Image Processing Techniques," *Optical Engineering*, vol. 21, pp. 432-435, 1982.
- [42] H. E. Cline, W. E. Lorensen, and A. S. Holik, "Automatic Moire Contouring," *Applied Optics*, vol. 23, pp. 1454-1459, 1984.
- [43] T. Yatagai, Inaba, and M. Suzuki, "Automatic Flatness Tester For Very Large Scale Integrated Circuit Wafers," *Optical Engineering*, vol. 23(4), pp. 401-405, 1984.
- [44] J. D. Gibson, *Principles of Digital and Analog Communications*, Collier Macmillan Publishers, London, 1989.
- [45] K. H. Womack, "Interferometric Phase Measurement Using Spatial Synchronous Detection," *Optical Engineering*, vol. 23(4), pp. 391-395, 1984.
- [46] W. W. Macy, Jr., "Two-dimensional Fringe-pattern Analysis", *Applied Optics*, Vol. 22, No. 23, 1983.
- [47] S. Wolfram, *Physica* 10D, pp. vii, 1984.
- [48] C. Buckberry, J. Davies, "Digital Phase-Shifting Interferometry and its Application to Automotive Structures", *Applied Optics Digest*, 17th-20th September, pp. 275-276, 1990.
- [49] A. Spik, D. W. Robinson, "Investigation of the Cellular Automata Method for Phase Unwrapping and its Implementation on an Array Processor", *Optics and Lasers in Engineering*, Vol. 14, No. 1, pp.25-37, 1991.
- [50] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "A Quasi Heterodyne Holographic Technique and Automatic Algorithms for Phase Unwrapping", *SPIE Vol. 1163*, August 1989.

- [51] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Analysis of Holographic Fringe Data Using the Dual Reference Approach", *Optical Engineering*, Vol. 30 No. 4, pp. 452-460, April 1991.
- [52] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Automatic Interferogram Analysis Techniques Applied to Quasi-heterodyne Holography and ESPI", *Optics and Lasers in Engineering*, 14, pp. 239-281, 1991.
- [53] J. J. Gierloff, "Phase Unwrapping by Regions", pp. 2-9, *SPIE Vol. 818 Current Developments in Optical Engineering II* (1987).
- [54] R. J. Green, J. G. Walker, "Phase Unwrapping Using A Priori Knowledge About The Band Limits Of A Function", pp. 36-43, *SPIE Vol. 1010 Industrial Inspection*, 1988.

Chapter 3

The Minimum Spanning Tree Approach to Phase Unwrapping

3.1 Introduction

A fringe field is similar to a contour map. However, the surface represented is sometimes discontinuous, which leads to discontinuities in the contour map. For example a rapid phase change in the profile of an object, or the density of a flow field may cause the Nyquist limit of the device capturing the interferogram to be exceeded. This is known as aliasing. In addition to such discontinuity types, which can have broad influence in an interferogram, noise of various kinds, optical and electronic, will also be present.

Discontinuities normally have a limited spatial extent in the fringe field. They can therefore be isolated to small areas of the interferogram. The MSTT approach to phase unwrapping, at the tile level, addresses the problem posed by discontinuities which, although they may be localised, are of such a size that they are not detectable by making simply pixel level inspections.

The approach utilises the fundamental principle that adjoining areas of the phase map should produce the same solution along their common boundary, unless an error is present. By segmenting the field into small areas, solving these separately and testing the agreement hypothesis it is possible to distinguish areas of confusion.

The segmented field is re-assembled under a priority scheme. Segments which seem to have solved unambiguously are employed in advance of less consistent ones. The priority scheme is designed in such a way that it considers a variety of factors in deciding the unwrap path.

In Graph Theory [1, 2, 3], graphs are used to represent many types of problem. A classic example is that of the Traveling Salesman. That is, of finding the shortest tour for a traveling salesman so that they visit each of a set of cities exactly once, and then return to their initial city. The difficulty of this problem is illustrated by the immense number of possible tours: $\frac{(n-1)!}{2}$ for n cities. This number grows very quickly, in fact faster than any finite power of n . This means that the brute force approach (a search of all candidates) is limited to small problems.

The TSP problem is analogous to several of more practical importance, including circuit layout and wire placement, for example. Research continues into algorithms which are able to produce suboptimal, but tolerable solutions in polynomial time. A promising approach, that of applying Neural networks, is first given in [4].

The phase unwrapping problem is not an analogue of the TSP. It is far more tractable. A sequential algorithm exists for the minimisation upon which it relies, with a time complexity of $O(n^2)$, where n is the number of nodes. A node may be either a tile or a pixel, depending upon whether regional or pixel level unwrapping is considered.

The graph provides a structure in which a number of factors may be weighed. For a connected undirected weighted graph describing the unwrapping problem, a Minimum Spanning Tree may be used as a mechanism for comparing alternative phase unwrapping routes.

The approach is employed hierarchically. Paths between pixels are compared to circumvent spike noise, and at a higher level tiles of pixels are compared to circumvent the larger scale discontinuities.

3.1.1 The Processing Objective

The aim of processing is to automatically produce a displacement map of the surface described by the phase fringe field. The processed output is to

contain a displacement value for each pixel of the captured field relative to some single origin.

3.1.2 The Processing Problem

In order to produce an automatic processing system, techniques must be developed to cope with the problems posed by the general fringe field.

For example, the sources of error in a Holographic interferogram, evaluated by the Quasi-Heterodyne technique are described in detail by Thalman and Dandliker [5]. They summarise the sources of error as follows:

- i) Fluctuations of the interference phase due to mechanical and thermal perturbations.
- ii) Inaccuracy of the phase steps.
- iii) The addition of interference patterns due to the overlapping of the cross-reconstructions.
- iv) Speckle noise.
- v) Electronic noise of the detector (CCD camera).
- vi) Non-linearity of the detector (CCD camera).

The errors above may be quantified. Thalman and Dandliker give an estimate of 1/100 of a fringe in total phase error for their RCA "Ultron" camera, which has a resolution of about 300 by 300 pixels. An error in the calculation of phase does not mean that the fringe field is necessarily discontinuous. That is the data may form a continuous map, with spike noise, even though the phase values themselves are in error. The unwrapping algorithm may therefore be applied to such maps, and the error in phase corrected once continuous data has been obtained. The following present serious problems to a phase unwrapping procedure, unless they are identified.

- i) Points of low modulation, which are produced from quantisation of intensity measurement. Large phase errors result at such positions.

- ii) Discontinuities, resulting from gaps in the fringe field or at the edges of objects.
- iii) Aliased areas, resulting from an insufficient pixel resolution to define all of the fringes. This is related to the complexity of the signal and the magnification of the field.

Any combination of the above may occur in a fringe field, complicating the unwrapping procedure, which requires a contiguous path across the field.

3.2 Tests for Low Modulation and Absence of Carrier

The low modulation test is widespread in fringe analysis. In Chapter Two it was seen that in a phase stepping system, with three fringe fields at a phase step of α , the expression below may be employed to compute the phase ϕ at a given pixel [5]

$$\phi = \arctan \left[\frac{(I_3 - I_2) \cos \alpha + (I_1 - I_3) \cos 2\alpha + (I_2 - I_1) \cos 3\alpha}{(I_3 - I_2) \sin \alpha + (I_1 - I_3) \sin 2\alpha + (I_2 - I_1) \sin 3\alpha} \right] \quad (3.1)$$

where I_1 , I_2 and I_3 are the intensities of the interferograms at the three phase positions α , 2α and 3α respectively.

It was also seen that in a system employing the Fourier Transform Technique, the phase is computed from an expression of the form

$$\phi(x, y) = \arctan \left[\frac{\Im[c(x, y)]}{\Re[c(x, y)]} \right] \quad (3.2)$$

where $c(x, y)$ is found as described in Chapter Two.

A low modulation point is identified when, for the quantity under the arctan, both the numerator and denominator are small. The probability of error in ϕ is then high.

In a Phase Stepping system, this is a successful method of detecting bad data points. The critical size for the numerator and denominator is empirically determined.

The test is not so successful for the FFT technique. This assumes that carrier fringes are present, modulated by the signal, in all areas of the interferogram. However, this may not be the case. For example an object with fringes projected on to it may be set on a neutral background, or experimental apparatus may cast shadows across the field.

The low modulation test may be applied to the FFT method to detect bad data points, but it is not so effective as when applied to the Phase Stepping method. This is explained as follows. After the side lobe is shifted (see FFT Method description in Chapter 2), the carrier is eliminated from the areas of valid data (which were previously modulated by the carrier). However, in other areas the carrier is introduced. The fringes in these areas are corrupted by any previous windowing type operations, applied whilst in the Fourier domain. This corruption permits the low modulation test to detect a percentage of bad data points in such areas. See Figure 3.24 for low modulation points detected during an FFT analysis. This effect is also illustrated in Section 3.3.3.

Aliased carrier fringes appear in areas of the field which lacked the carrier in the original interferogram. These areas may be detected in advance of the FFT process by examining the interferogram to locate areas where fringes are absent. A thresholding strategy may then be employed. The percentage of points above an empirically derived intensity level, in a given area, is used to determine whether any fringes are present. This strategy has been implemented along with the low modulation test, the areas examined correspond to those of the tiles used in phase unwrapping. Any tiles covering an area lacking the carrier are not processed.

As another alternative, Donovan et al. [6] have instead introduced the carrier into these areas. The areas are again detected by a thresholding strategy. This strategy has the advantage that the DC term in the Fourier domain is reduced, as the transform of the area lacking fringes is not then superposed on the rest of the transform. In order to partially eliminate the DC term the transform of the area lacking the fringes may be computed separately and subtracted in the Fourier domain, this approach is applied by BAe (Chapter 5) and is similar to Kreis's approach (Chapter 2).

3.3 Examples of Discontinuity Types

3.3.1 Aliasing Induced

The discontinuity shown in Figure 3.1 and 3.2 is caused by insufficient photographic resolution. A shock has been induced. The sudden increase in fringe density has not been accommodated by the spatial resolution of the film. The image has subsequently been digitised. It can be seen that the profiles of the fringes in the wrapped phase map of Figure 3.2 are disrupted. This has caused fringes to terminate suddenly. Such fringe terminations are detected in the phase unwrapping scheme, this is discussed later in the chapter. Figure 3.2 shows the wrapped phase map after the FFT analysis method has been applied.

3.3.2 Hole in Object

Figures 3.3 and 3.4 give an example of an interferogram in which the subject contained a hole. The hole was produced by a spark plug. The source images from which this example has been computed were kindly supplied by Jeremy Davies and Clive Buckberry of Rover's Research facility at Gaydon. The experiment is described in reference [11]. The phase stepping method has been used to generate the wrapped phase map, in an Electronic Speckle Pattern system. The low modulation test successfully defined many of the problem areas in this example, including the spark plug hole, see centre of Figure 3.34.

3.3.3 Absence of Carrier

Figures 3.5 and 3.6 show the effect on the wrapped phase map when an area of the interferogram lacks the carrier in processing by the FFT method. The effect can be eliminated by one of the methods mentioned in Section 3.2.

3.4 Adoption Of The MST Tiling Method

It is difficult to design an algorithm to unwrap a general fringe field when one considers the field as an indivisible unit. It is unrealistic to apply a single

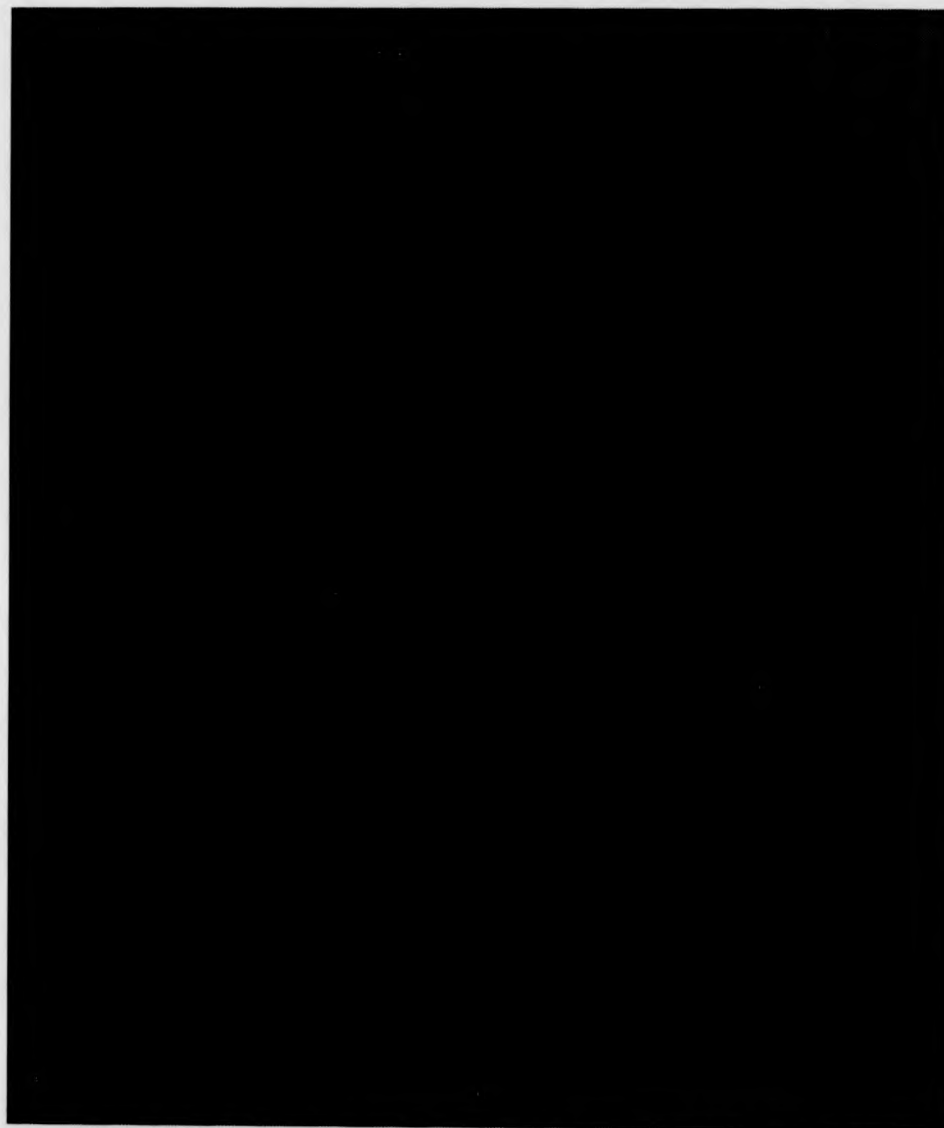


Figure 3.1: Example of Aliasing Induced Discontinuity, (Original Interferogram)

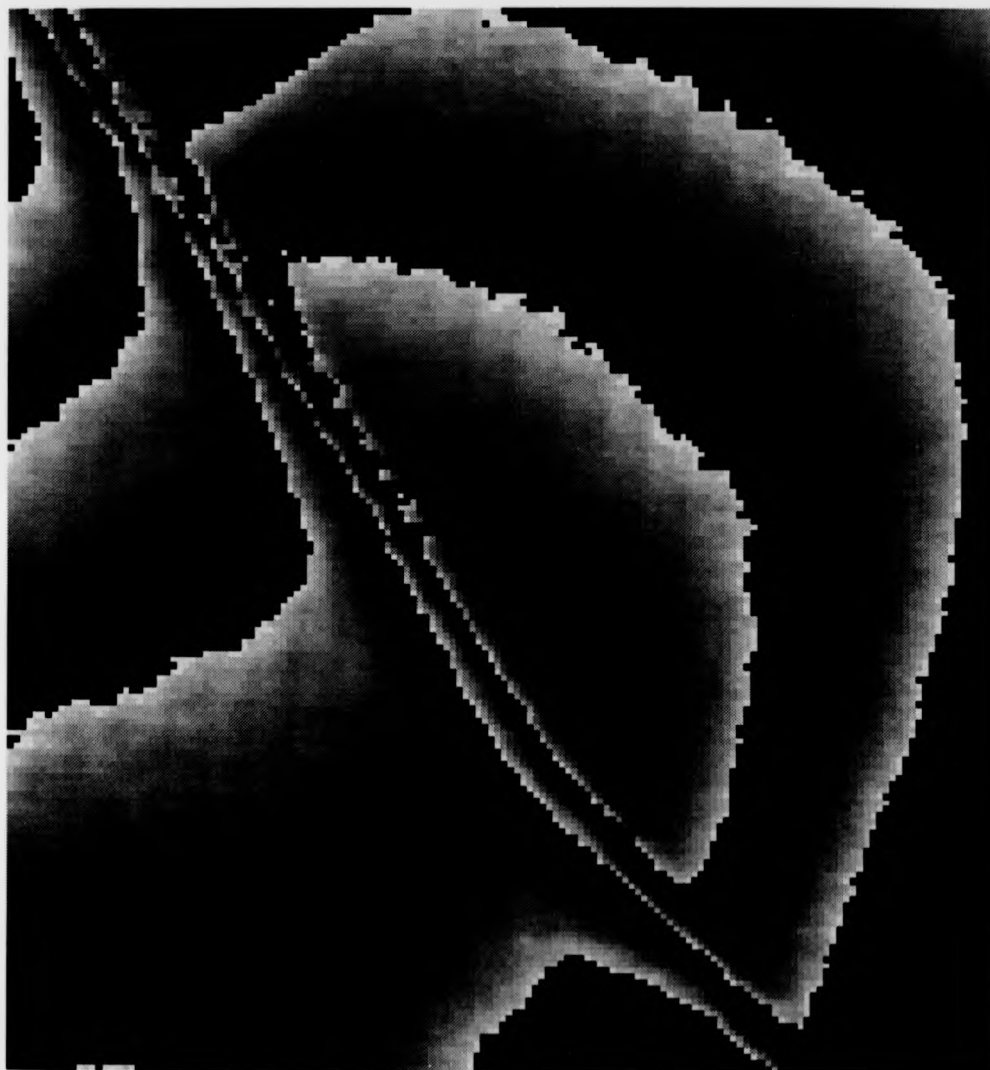


Figure 3.2: Example of Aliasing Induced Discontinuity, (Wrapped Phase Map by FFT Method)



Figure 3.3: Example of Discontinuity Introduced by Hole in Object, (Electronic Speckle Interferogram after Prefiltering)



Figure 3.4: Example of Discontinuity Introduced by Hole in Object,
(Wrapped Phase Map by Phase Stepping with 3 Images)



Figure 3.5: Example of Discontinuity Introduced by Absence of Carrier,
(Original Interferogram)



Figure 3.6: Example of Discontinuity Introduced by Absence of Carrier, (Wrapped Phase Map by FFT Method)

strategy and expect problems such as those outlined above to be resolved.

As was seen in Chapter 2, techniques which deal with spike noise can be applied with success in some circumstances. However, the downfall of such algorithms results when the field is not broadly continuous. The algorithms considered in Chapter 2 lack strategies for dealing with discontinuities that do not show up through pixel comparisons. For example, Huntley's algorithm compares the consistency of phase data along paths of pixel width. Discontinuities which are significantly larger than the pixel, say 3 or more pixels wide, may appear to have consistent pixel width phase paths through them. These types of discontinuity are therefore invisible to such strategies.

3.5 Graphs in Graph Theory

A graph $G = (V, E)$ consists of a set of objects $V = \{v_1, v_2, \dots\}$ called vertices, and another set $E = \{e_1, e_2, \dots\}$, whose elements are called edges, such that each edge e_k is identified with an unordered pair (v_i, v_j) of vertices. The vertices v_i, v_j associated with edge e_k are called the end vertices of e_k . The most common representation of a graph is by means of a diagram, in which the vertices are represented as points and each edge as a line segment joining its end vertices. Often this diagram itself is referred to as the graph [2]. Figure 3.7 shows an example of a graph.

3.5.1 The Concept of Connectedness

A graph is connected if it is possible to reach any vertex from any other vertex by traveling along the edges.

A graph G is said to be connected if there is at least one path between every pair of vertices in G . Otherwise, G is disconnected. Figure 3.7 shows a connected graph.

A disconnected graph consists of two or more connected graphs. Each of these connected subgraphs is called a component. Figure 3.8 shows a disconnected graph with two components.

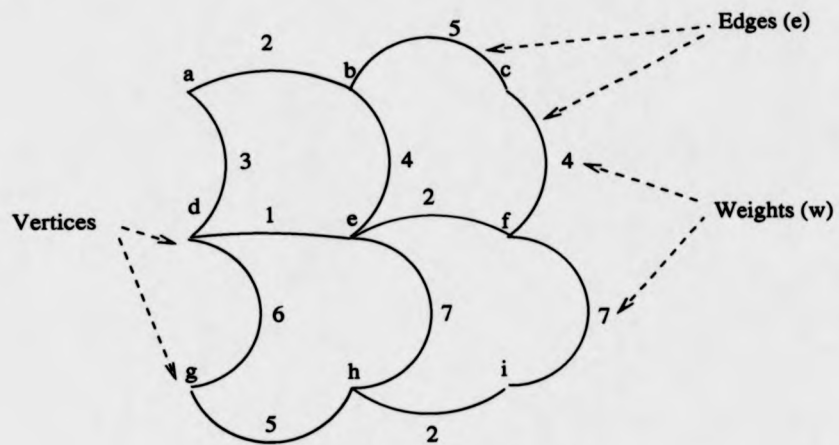


Figure 3.7: Example of a Graph

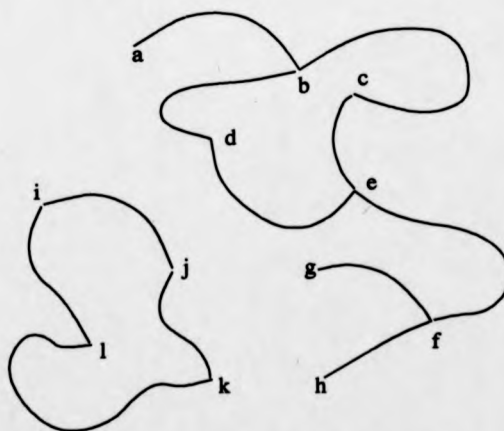


Figure 3.8: A Disconnected Graph

3.5.2 Trees

A tree is a connected graph without any circuits. Figure 3.9 is a tree. Some properties of trees are outlined below.

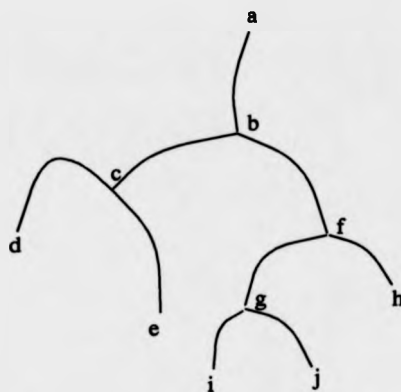


Figure 3.9: An Example of a Tree

There is one and only one path between every pair of vertices in a tree, T .

If in a graph G there is one and only one path between every pair of vertices, G is a tree.

A tree with n vertices has $n - 1$ edges.

3.5.3 Minimum Spanning Trees

A spanning tree in a graph G is a minimal subgraph connecting all the vertices of G . If graph G is a weighted graph (i.e., if there is a real number associated with each edge of G), then the weight of a spanning tree T of G is defined as the sum of the weights of all the branches in T . Among all of the spanning trees of G , those with the smallest weight are called Minimum Spanning Trees. Figure 3.10 shows the minimum spanning tree of the graph in Figure 3.7. [2]

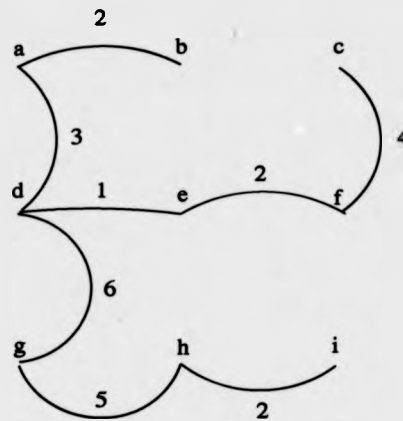


Figure 3.10: A Minimum Spanning Tree of the Weighted Graph

3.6 Hierarchical Phase Unwrapping Using Minimum Spanning Trees

The phase unwrapping method is divided into two levels. The lowest level covers the procedures required to unwrap phase pixel to pixel, within tiles. The highest level considers the assembly of the tiles into a whole field solution, the weighting strategies employed etc.

The strategy for connecting the solved tiles is examined first. This is easier to digest as the concepts are more high level. In operation, however, the low level processes are applied first. As a product of the low level processing the factors required at the higher level are extracted.

3.7 High Level Phase Unwrapping, Connection of Unwrapped Tiles

Figure 3.11 shows a tiled section of the fringe field. Firstly, each tile is considered to be a vertex in a weighted connected graph G , Figure 3.7. Edges are added to the graph where tiles have a common boundary. That is each tile vertex is connected, by edges e , to the vertices of neighbouring tiles.

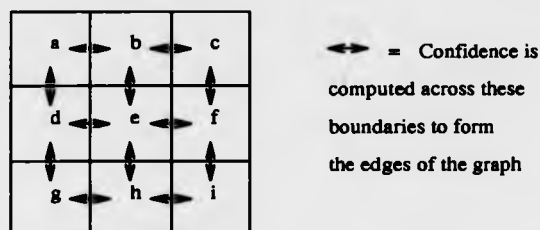


Figure 3.11: Tiled Section of the Fringe Field Corresponding to the Weighted Graph

The weights w , corresponding to the edges, are calculated, from a variety of factors, to represent the validity of phase unwrapping across the boundaries.

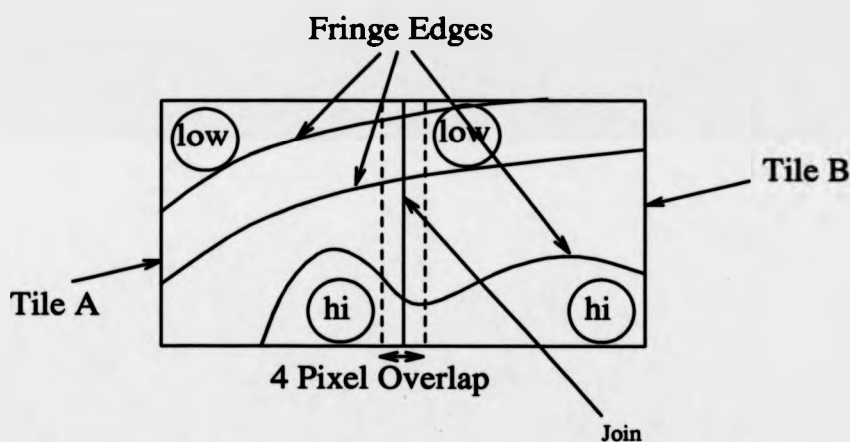


Figure 3.12: Neighbouring Tiles Showing Overlap

3.7.1 Factors for Assessing the Quality of Data in the Field at the Tile Level

3.7.1.1 Agreement of Neighbouring Tiles

The first factor in the weighting quantifies the fit of the solutions in adjacent tiles. This is calculated from a comparison of the profiles of the unwrapped solutions at the spatial boundaries of the tiles. The tiles overlap one another to facilitate this comparison, See Figure 3.12.

The unwrapped solution at the boundary of each tile is dependent upon the data within the tile. That is, bad data within the tile is likely to affect the solution obtained at its edge. Therefore a good tile, encompassing valid data, placed next to a bad tile, encompassing invalid data, will produce a bad fit when the profiles of the solutions are compared, at the common boundary.

3.7.1.2 Points of Low Modulation

Points of low modulation are identified as described in Section 3.2. They are used as an indicator for bad data. The low modulation factor is computed from the sum of the number of low modulation points found on either side of the tile boundaries. This is a very effective measure, particularly in Phase Stepping systems.

3.7.1.3 Fringe Density

The third factor employed is fringe density. It is suggested that in general the more fringes there are in a given area of an interferogram, the more probable an error is likely to result. The number of fringes is related to the complexity of the signal. The more complex the signal, the greater the likelihood of error.

However, there is a qualification. It has been noted that where aliasing is present, that is where the Nyquist limit of the device capturing the interferogram has been exceeded, the number of fringes appears to decrease. In such cases this test is unreliable. However, both the neighbourhood agreement and fringe termination tests are successful in such areas.

The density of fringes has been estimated using the edge detection procedure. That is the number of points found on fringe edges provides an estimate of fringe density. The density factor used in the weighting strategy represents the condition at the boundary between tiles. It is therefore computed from the sum of the densities on either side of the boundary.

3.7.1.4 Fringe Edge Termination Points

The last factor currently employed in the weighting, is obtained from a test for points where fringe edges terminate, See Figure 3.13. That is where otherwise continuous fringes suddenly stop. Such points can occur for a

number of reasons. For example, at the edges of objects, or as a result of aliasing.

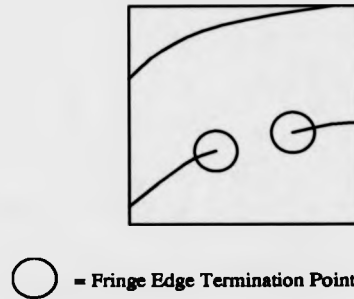


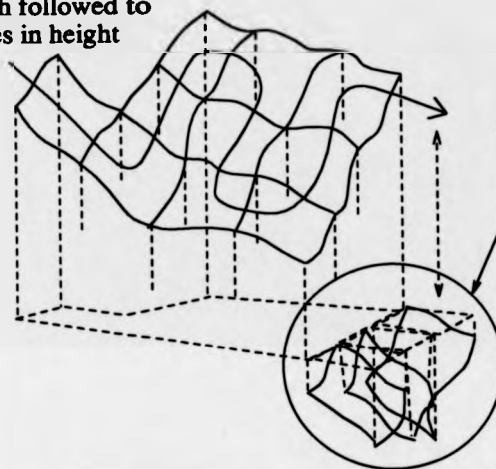
Figure 3.13: Edge Termination Points

A function is computed from the factors discussed above. A more detailed discussion of this function is given later. It is sufficient to mention at this point that a low value for the function is defined to represent a good route, and a high value a bad one. The assembly path is then obtained by constructing the Minimum Spanning Tree of the connected undirected graph formed by the tile vertices. Prim's algorithm is used to construct this tree [8].

The tiles are repositioned in height as the tree is computed, see Figure 3.14. The height offset of each new tile added is calculated by summing height across the tiles, from the root tile to the edge connecting the new tile to the tree. Defective tiles are forced to the tips (or leaves) of the tree branches and so distortions are reduced.

The strategy is analogous to the solution of a jigsaw puzzle. The solution begins with a single piece (the root tile). New pieces are then added successively around it until the complete picture is made. Each tile may be thought of as a jigsaw puzzle piece that must be fitted to form the completed fringe field. At each stage the piece that is thought to fit best is added. The solved tiles have edge profiles that must be matched, but, unlike the jigsaw it is not guaranteed that profiles match exactly.

Tree branch followed to
arrange tiles in height



These tiles
will be added
next

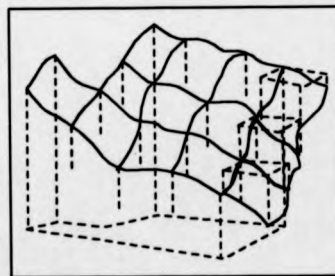


Figure 3.14: Tiles being Arranged at their Correct Height Offsets

3.7.1.5 Metrics

As outlined above, to produce a solution to the phase unwrapping problem a path across the field has be found which minimises the effect of errors in the interferogram. Parameters may be extracted from the fringe field to aid in this minimisation process. The factors, at present, include

- i) The agreement of solutions in neighbouring areas.
- ii) The extent of low modulation noise.
- iii) The local fringe density.
- iv) The extent of fringe terminations.

Some sort of mathematical framework is required to quantify and compare the relative demerits of these factors.

Suppose there are an arbitrary number, R , of such factors to be compared. These may be represented in a space of R dimensions, each factor on a different axis.

To facilitate the comparison it becomes necessary to define a confidence measure metric between points in this R dimensional space.

In order for any such metric to be valid a metric space must exist. The following conditions must be satisfied for this to be the case (suppose x , y and z are points in the space).

- i) $d(x, y) = d(y, x)$
- ii) $d(x, y) \leq d(y, z) + d(x, z)$
- iii) $d(x, y) \geq 0$
- iv) $d(x, y) = 0$ iff $y = x$

Such a metric is often referred to as a distance function.

However, even with the existence of the metric space, it is still important to realise that variables with differing statistical characteristics cannot be directly compared. A normalisation process needs to be applied to each factor to bring them into a domain where they may be directly compared.

The dynamic ranges of the various axes are forced to be well behaved. An overall figure of confidence, for a given step in the unwrapping route, may then be computed.[9]

Suppose that there are M weighting factors. Suppose that the distribution of a given weighting factor is W_j , where $(j = 1, 2, \dots, M)$, is used to distinguish between the M factors. Further suppose that $W_{j1}, W_{j2}, \dots, W_{jN}$ are samples of W_j found across the boundaries of adjacent tiles, where N is the number of boundaries. The normalised variables $w_j(j = 1, 2, \dots, M)$ are obtained from Equation 3.3 below

$$w_{ji} = \frac{W_{ji} - \bar{W}_j}{s_{W_{jN-1}}} \quad (3.3)$$

For $(i = 1, 2, \dots, N)$, and $(j = 1, 2, \dots, M)$ where $s_{W_{jN-1}}$ is the standard deviation of the sample of W_j .

The combined weighting factor w_c is obtained from the mean of the weighting factors after normalisation

$$w_{ci} = \frac{\sum_{j=1}^M w_{ji}}{M} \quad (3.4)$$

The normalisation process is intended to bring the weighting factors into a domain where they might be directly compared. There are alternatives to the strategy adopted. One of the problems with the above approach is that, in the situation where a weighting factor has a rather uniform distribution, the minor variations from sample to sample may be amplified unnecessarily in the normalisation process and so reduce the effectiveness of the comparison tests.

One way of combating this problem would be to record an analysis history for similar images. In this way the weighting factors of significance in a particular experiment could be isolated. Those factors which did not seem to contribute could then be ignored. Such factors simply add noise to the weighing procedure, under the system described above.

3.7.2 The Minimum Spanning Tree Algorithm

Figure 3.1 shows a tabular representation of the graph G . The starting

Vertex	\uparrow	\downarrow	\leftarrow	\Rightarrow
a	- ∞	d 3	- ∞	b 2
b	- ∞	e 4	a 2	c 5
c	- ∞	f 4	b 5	- ∞
d	a 3	g 6	- ∞	e 1
e	b 4	h 7	d 1	f 2
f	c 4	i 7	e 2	- ∞
g	d 6	- ∞	- ∞	h 5
h	e 7	- ∞	g 5	i 2
i	f 7	- ∞	h 2	- ∞

Table 3.1: Tabular Representation of the Weighted Graph

vertex (root of the tree) is located by descending the table and selecting the vertex (corresponding to tile or pixel) from those with the highest degree (number of edges), and the lowest sum of edge weights. Some tiles may not have 4 neighbours as bad tiles may have already been deleted from the graph. The selection of this vertex is not essential for construction of the tree, but it places the root at the most confident pixel or tile and makes construction of the tree easier to follow.

The vertices are denoted by $v_{\text{row index in table}}$. Let r be the row at which the root vertex is located. The root is then v_r . Prim's algorithm then proceeds as follows. Connect vertex v_r to its nearest neighbour, the vertex with the smallest weight entry in row r of the table, v_k . Then consider v_r and v_k as one subgraph, and connect this subgraph to its closest neighbour (i.e. to a vertex other than v_r and v_k that has the smallest entry among all the entries in rows r and k). Let this new vertex be v_i . Next regard the tree with vertices v_r , v_k , and v_i as one subgraph, and continue the process until all n vertices have been connected by $n - 1$ edges [2, 8].

3.7.3 Selection of an Appropriate Tile Size

The confidence calculation is such that there is a relationship between the best tile size, the density of fringes and the size of breaks in fringe edges. The ideas behind this are explored below.

Figure 3.15 shows the plan view of a tile. A single disrupted fringe edge is shown. For simplicity a fringe counting algorithm is considered, working from the top edge of the tile to the bottom. That is, the tile is to be unwrapped by a series of vertical scans, indicated by l , m and n in the Figure. The same reasoning applies to the Minimum Spanning Tree approach for the pixel level. The algorithm is to decide whether the fringe edge is truly present, or whether the apparent edge points are due to noise. This is of particular importance for the connection of the tile neighbour along the line PQ . That is the height offset of the tile which is to connect along PQ , with respect to the tile considered, relies upon the correct solution of this problem and so the tile as a whole.

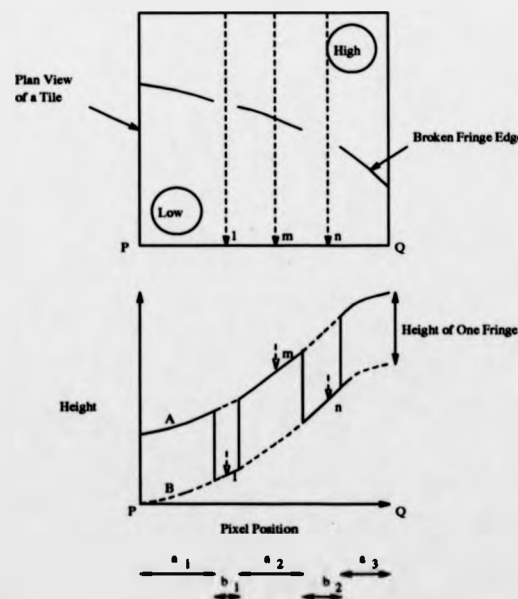


Figure 3.15: Selection of Tile Size: Tile with One Broken Fringe Edge

Figure 3.15 also shows the height profile at the bottom edge of the tile,

along the line PQ . This solution contains steps, as a result of some of the fringe counting scans having missed the fringe edge, or alternatively having struck an erroneous fringe edge. Consider the two possibilities, which differ by one fringe in height. The first curve A represents the cases where a fringe edge has been detected, and the second curve B represents those where no edge has been detected. The strategy employed, to determine which solution is the most likely to be correct, involves comparing the lengths of the curve sections with respect to the horizontal axis. In this case, for example, the sum for the A solution is given by

$$l_A = a_1 + a_2 + a_3 \quad (3.5)$$

and for the B solution by

$$l_B = b_1 + b_2 \quad (3.6)$$

If l_A is greater than l_B then the A solution is selected, otherwise the B solution is chosen.

l for the various possibilities is actually computed by comparing the PQ profile with the complementary solution obtained from the adjacent tile. To be exact, the two solutions from the adjacent tiles are subtracted from one another, to yield a list of differences. These differences are then sorted. Similar differences are grouped during the sorting process. Suppose that there are M pixels between P and Q and that there are N unique values for difference, the unique values being denoted as d_i ($i = 1, \dots, N$). Initially these values form an unsorted list. After sorting the l_i ($i = 1, \dots, N$) are obtained by computing the length of each sublist, all the elements of each sublist being identical. The difference which occurs the most often, d_k given by the length of the longest sublist l_k , is used as the most probable solution. The difference d_k , permits the two adjacent tiles to be brought into registration with one another. l_k , after some normalisation, is used as the 'neighbour agreement' factor in the weighting strategy for connecting tiles together. In fact two rows of pixels are considered in the manner described in the current implementation.

A choice of only two possibilities exists in the example above, and so the

strategy works effectively. The idea in the selection of tile size is to obtain this position. The confidence with which such a conflict may be resolved, if more broken fringes cross the tile, decreases in a non-linear fashion.

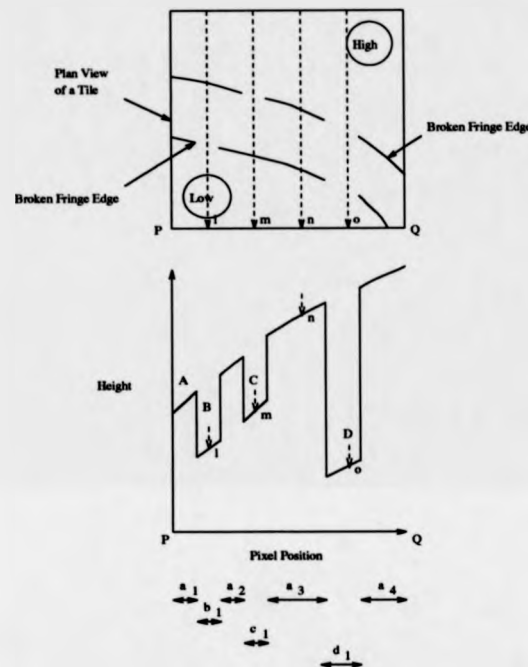


Figure 3.16: Selection of Tile Size: Tile with Two Broken Fringe Edges

Consider Figure 3.16, where two disrupted fringes cross a tile. There are then four possible solutions, *A*, *B*, *C* and *D*. The addition of one more fringe edge has doubled the number of solutions that must be considered. As the range of the solutions is greater, so it is more confusing for the algorithm as a binary choice no longer exists (*A* or *B*).

If f is the actual number of fringe edges which cross a tile, then the number of solutions which must be distinguished between, N , in the worst case, increases as a power of two;

$$N = 2^f \quad (3.7)$$

The discussion above deals with one extreme of the tile size issue, the

upper limit. The other extreme, that is the question of how small tiles should be, must also be addressed.

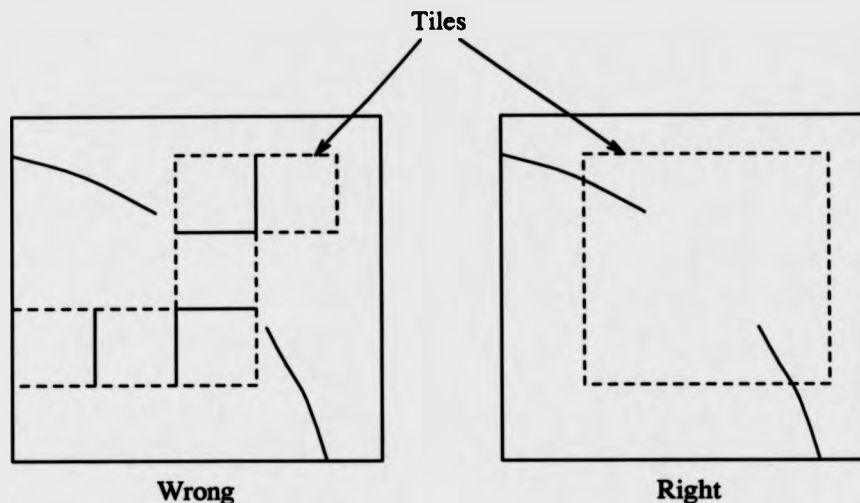


Figure 3.17: Illustration of Effect of Tile Size

If the tiles are smaller than the fringe spacing, the connection flow will tend to track the fringes. However, in this case the confidence calculation for a tile bridging the area of a broken fringe edge, may give a falsely high confidence level where adjacent fringes appear to merge, see Figure 3.17 this would cause a ridge to be formed along the length of the edge common to the fringe pair. That is, the broken area would be forced to the same height in the two fringes. This situation may be avoided by placing a lower limit on the tile size, such that the size exceeds that of the largest break in any fringe edge of the field.

There are two points to consider, then, in the selection of tile size. The first point being that the tiles should not be so large that many edges fall across them, as a multitude of possible solutions confuses the unwrapping process. The second point being that the tiles should not be so small that they often fall between gaps in broken edges and so cause adjacent fringes to be merged.

3.8 Low Level Phase Unwrapping, Pixel to Pixel

The Minimum Spanning Tree approach to unwrapping phase at the pixel level has developed from a fringe counting method. The initial procedures of both methods include the steps outlined below

- i) Computation of wrapped phase map. There are at least two techniques for computing the wrapped phase map. The Phase Stepping and FFT Methods are both described in Chapter Two.
- ii) Fringe edge detection. An adaptive thresholding technique applied to the Sobel operator is employed here. This is described in detail in Chapter 4.
- iii) The unwrap of phase, tile row, by tile row.
- iv) The unwrap of phase, tile column, by tile column.

At this point the fringe counting and minimum spanning tree techniques divide.

In the case of fringe counting, one set of scans (either vertical or horizontal) forms the basis of the solution whilst a single scan from the orthogonal set is selected to arrange the elements of the first set relative to one another [10].

In the minimum spanning tree case, every pixel is considered to be a vertex in a graph of confidence over the tile. The problem is then, as in the case of the tile assembly, to construct an unwrapping path which maximises confidence. The weights of graph edges are again used to signal the confidence of a particular pixel to pixel route.

Each pixel has four neighbours (apart from those at the edge of the field), and these will be referred to by the compass points, north, south, east and west. In a similar fashion to the tiles, there is one edge in the graph to relate the confidence of unwrapping across each pixel neighbour pair. The weight of each edge is calculated from the rate at which phase changes, denoted by a difference, in stepping between each pixel pair. The minimum spanning

tree, therefore, minimises pixel to pixel phase changes in the phase unwrap path.

Spike noise may be characterised by a rapid change in phase, between the spike and surrounding pixels. By unwrapping phase via a path (tree) which minimises the phase change at each pixel to pixel step, then noise points are prevented from entering the solution until the outer leaves of the tree are reached, where they no longer threaten the solution. The minimum spanning tree approach therefore presents a noise immune phase unwrapping strategy.

3.8.1 The Calculation of Edge Weights at the Pixel Level

The edge weights may be calculated from the horizontal and vertical tile scans, obtained above. These contain partially unwrapped data, localised to either rows or columns. The local nature of the data means that the absolute offset of the data relative to the rest of the field is unknown. However, for the purpose of weight computation only relative phase changes are necessary.

The question really being asked when each edge weight is computed is, *How smooth would the unwrapped pixel to pixel phase change be, if unwrapping took place through this pixel pair, compared to one of the other possibilities?* There is an implicit assumption that the fringe field represents a continuous function that is changing relatively slowly. That is, the signal does not, in the main, contain high frequency components that could be confused with noise. Each pixel effectively represents a crossroads at which the best direction to move, in order to minimise phase unwrapping errors, must be determined. The vertical scan set is used to weigh the merits of the north/south options, whilst the horizontal scan set is used to weigh the merits of the east/west options.

The steps in the unwrapping procedure continue

- i) Computation of east/west pixel edge weights.
- ii) Computation of north/south pixel edge weights.
- iii) Phase unwrap during calculation of minimum spanning tree.

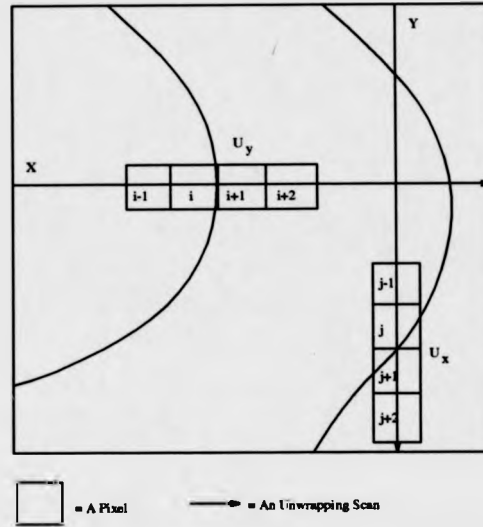


Figure 3.18: Pixel Level, Computation of Edge Weights. Weights are Computed over Four Pixels for Each Pair

The edge weights are actually computed as an average over a two pixel area, see Figure 3.18. The effect of an *averaging* filter, placed over a spike noise point, is to spread the spike so that it raises the values of surrounding pixels. Using the average in the graph weighting strategy serves to increase the weight of graph edges which connect to pixels surrounding the noise point. These are embedded in the unwrapping path *before* the noise spike is actually reached, thereby causing the algorithm to delay further its approach to the noise spike.

Referring to Figure 3.18, X and Y represent a horizontal and vertical unwrap scan, respectively. Let the unwrapped phase of row scan y be denoted by $U_y(i)$, where i is an index to the pixels of the scan, and similarly let $U_x(j)$ represent the unwrapped phase of column x at pixel j of the scan. For a row pixel pair i and $i + 1$, the weight of the edge connecting the pixel vertices of row y , is denoted by w_y and calculated via Equation 3.8 from the unwrapped rows

$$w_y = |(U_y(i) + U_y(i - 1)) - (U_y(i + 1) + U_y(i + 2))| \quad (3.8)$$

and similarly the north/south edge weights are calculated from the unwrapped columns in Equation 3.9

$$w_x = |(U_x(j) + U_x(j - 1)) - (U_x(j + 1) + U_x(j + 2))| \quad (3.9)$$

The change in phase between the unwrapped pixel pairs δ is also recorded in order that phase may be unwrapped during formation of the minimum spanning tree

$$\delta = U(i + 1) - U(i) \quad (3.10)$$

Figure 3.19 illustrates pixel level phase unwrapping using the analogy of a mountaineer. The climber sets out from a plateau, in this case the foot of the mountain. He follows the easiest path he can find, where the ground is even and rises easily. The ruggedness of the terrain is similar to noise in the interferogram. When confronted by steeply rising ground, or an obstacle he cannot climb, he retraces his route and takes a less steep path.

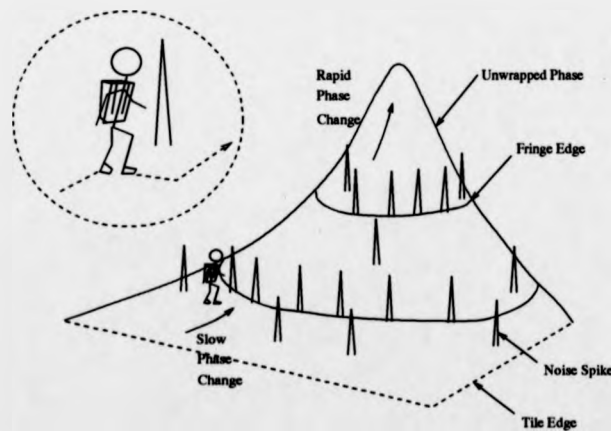


Figure 3.19: Mountaineer Analogy in Pixel Level Phase Unwrapping

3.8.2 The Interaction of Tile Level Phase Unwrapping with the Pixel Level

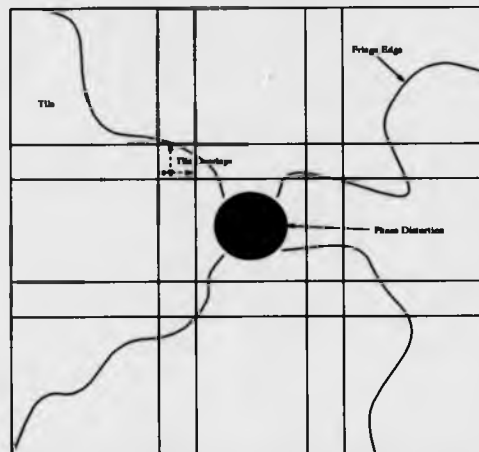


Figure 3.20: Interaction of Tile Level Phase Unwrapping with the Pixel Level

An example is given in Figure 3.20 of a phase discontinuity which is entirely contained within the central tile. The following series of figures indicate the manner by which this discontinuity is circumvented.

It can be seen that four fringe edges enter the central tile and terminate before reaching the centre. The pixel level phase unwrap of the central tile would, as described above, minimise the pixel to pixel phase change from pixel to pixel in the solution. The central area of distortion would contain larger pixel to pixel phase changes than the surrounding area of the tile and so, the pixel level unwrapping strategy would take a path which tended to circumnavigate the central area as shown by A in Figure 3.21. This would mean that there was no mismatch between the solutions in the overlap areas of neighbouring tiles. The discontinuity would be wholly contained within the tile and isolated by the pixel level strategy.

Imagine now that the same situation exists, but that the central area of the tile is perfectly smooth. That is the discontinuity is not detectable by the pixel level strategy as only small changes in phase occur across the discontinuity. Such a situation might occur from excessive filtering of a Speckle image, for example, Figure 3.22. In this case the comparative smoothness of the phase change in the centre of the central tile causes phase unwrapping to spread out from the centre towards the bounds of that tile. This

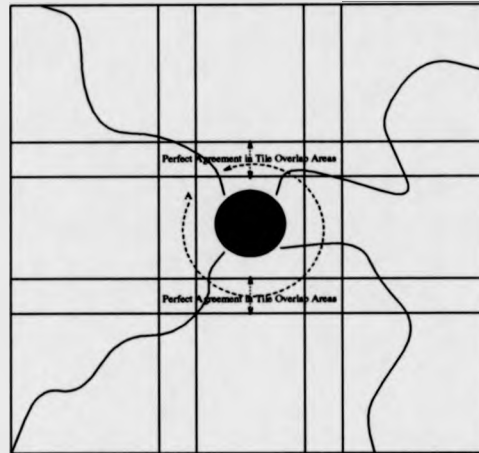


Figure 3.21: Pixel Level Phase Unwrap Circumnavigates Phase Distortion

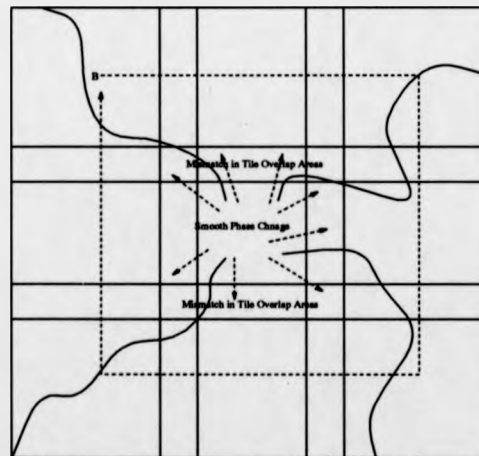


Figure 3.22: Tile Level Phase Unwrap Circumnavigates Smooth Area

in turn introduces phase jumps along the edges of the central tile, where some paths would have crossed fringe edges, and others not. The solutions for the surrounding tiles are not affected by the discontinuity in the central tile, and so a mismatch is detected in the tile overlap areas. This leads to increased weights on the graph at the tile level for the edges related to the central tile. In this situation, then, the tile level of phase unwrapping enables circumvention of the discontinuity as shown by B in Figure 3.22.

3.9 Examples of Tile Level Phase Unwrapping

A couple of examples are necessary at this point to illustrate the topics discussed above. Examples of FFT and Phase Step analysis are given.

The FFT example shows deformation of a metal disc. A full description of this experiment is given in Chapter 5.

The original images for the Phase Stepping example have been generated by a fibre-optic-based system, developed at the Rover group research centre, Gaydon (images presented by kind permission). They show a cylinder head chamber, the four valves can be made out in the image. The experiment is described in reference [7].

The discussion begins at the point where the wrapped phase maps have been computed.

3.9.1 FFT Example of Tile Level Phase Unwrapping

Figure 3.23 shows the wrapped phase map for the deformed metal disc of Figure 2.13, in the last Chapter. This image has been generated by the FFT method. There are several discontinuity types in this image. The most prominent discontinuities are due to the aliased carrier fringes at the top and bottom of the image. As was mentioned earlier there are methods for detecting these aliased areas based upon thresholding the original cosinusoidal interferogram and by detection of low modulation points, Figure 3.24.

An interesting discontinuity occurs at the surround of the metal disc. The disc is recessed by a few millimetres. A shadow is therefore cast by the

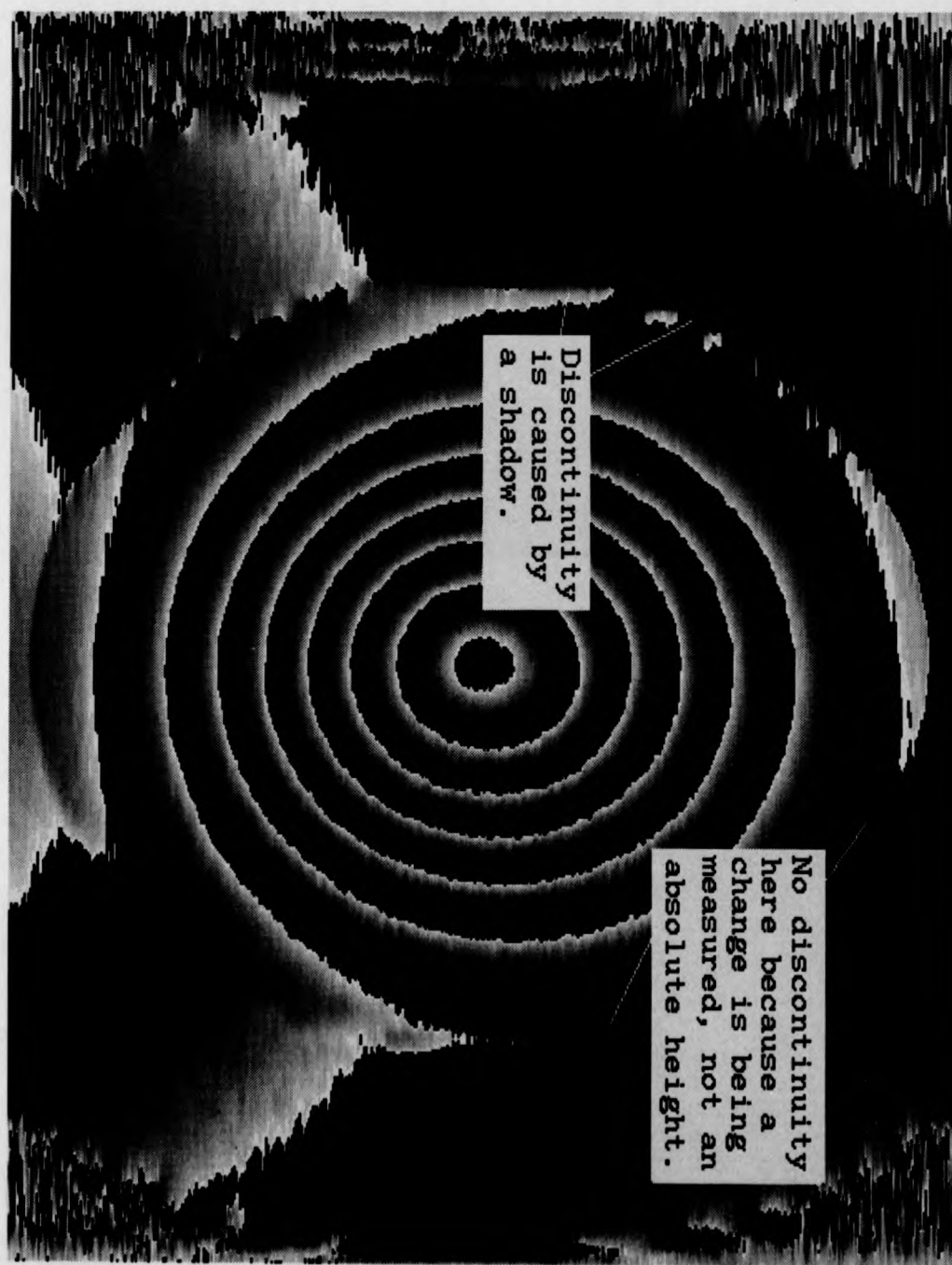


Figure 3.23: Wrapped Phase Map for Disc Showing Problem Areas

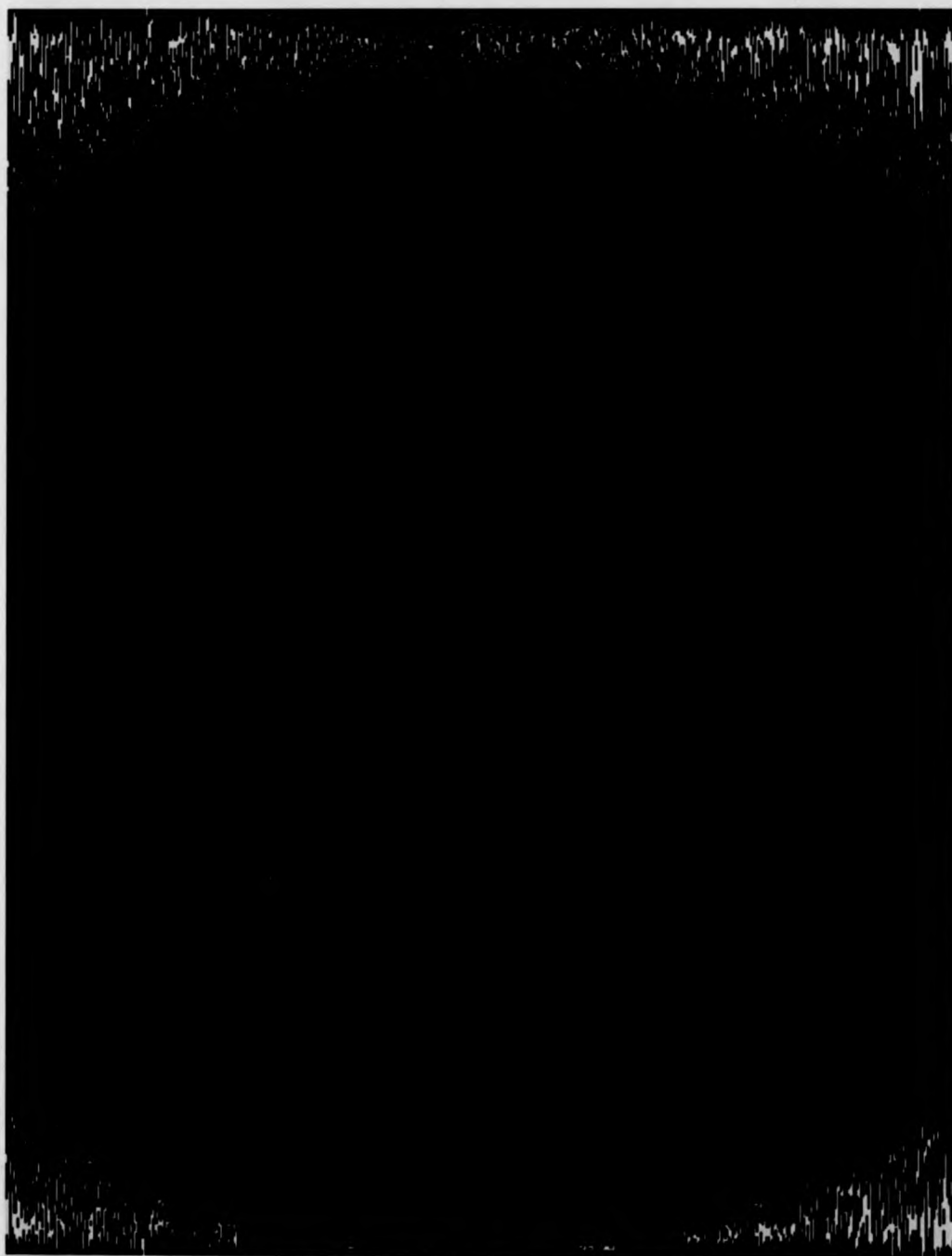


Figure 3.24: Points of Low Modulation for Disc (shown in white)

surround. There is a physical discontinuity between the disc and the surround due to the height difference. In the wrapped phase map, the discontinuity is visible in the area of the shadow, but not at all points along the rim of the disc. The deformation measurement is a difference measure, between the state before deformation and after. This explains why no discontinuity appears at the edge of the disc other than where the shadow is present. It will be seen that the unwrapping strategy detects the presence of the discontinuity caused by the shadow and solves the majority of the field in a consistent manner in spite of it. The tile size has been selected as 44 pixels, including a 4 pixel overlap. This is comparable to the fringe spacing upon the disc, and larger than the size of the breaks between fringe edges. The reasons underlying the selection of tile size are described in Section 3.7.3.

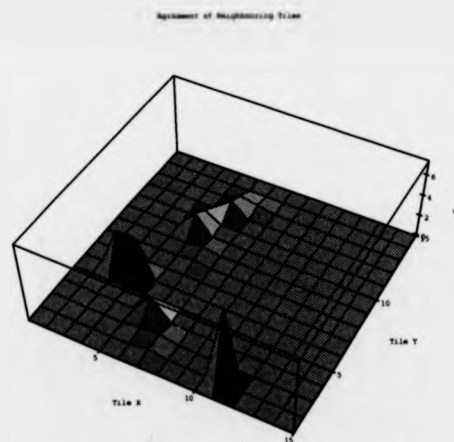


Figure 3.25: Weighting Factor for Disc : Agreement between Neighbouring Tiles

The series of Figures 3.25, 3.26, 3.27, 3.28 and 3.29 show mesh plots representing the various weighting factors described earlier in this Chapter and the combined weighting, which is a sum of the other factors, employed in construction of the tile level minimum spanning tree. The orientation of

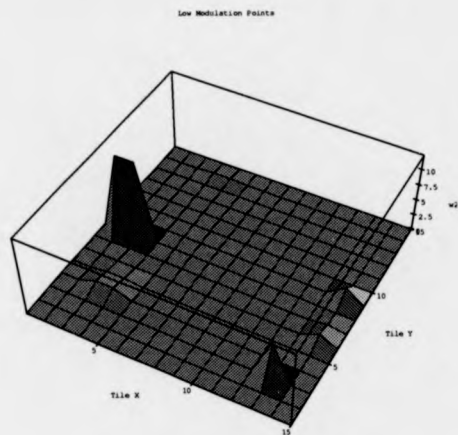


Figure 3.26: Weighting Factor for Disc : Low Modulation Points

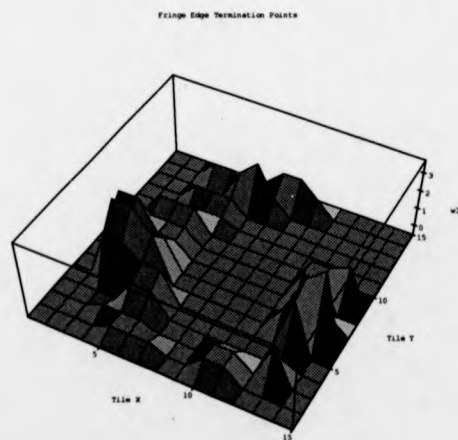


Figure 3.27: Weighting Factor for Disc : Fringe Edge Termination Points

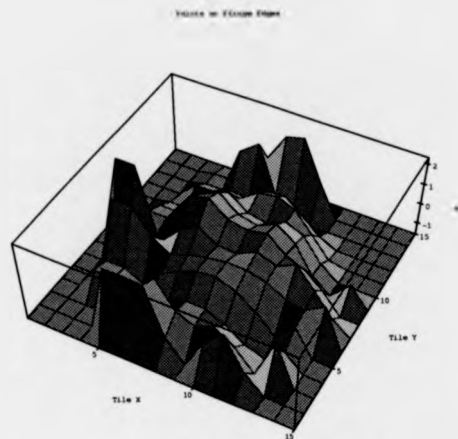


Figure 3.28: Weighting Factor for Disc : Points on Fringe Edges

these plots with respect to the disc image, shown in Figure 3.23, is such that the x axis of the mesh plot aligns with the vertical of Figure 3.23 and the y axis aligns with the horizontal. The origin for the mesh plots is in the top left hand corner of Figure 3.23.

Figure 3.25 shows the case for the agreement of neighbouring tiles. Each grid point represents the edge weight of the factor considered (in this case neighbour agreement). The greater the value of any factor, then the poorer the image quality at the referenced position. It should be noted that the aliased tiles, at the top and bottom edges of the image, have already been automatically eliminated at this stage of processing by the thresholding and low modulation tests described earlier in the chapter. This is the reason why there are no high peaks around the edge of the plot in Figure 3.25. In fact as the image is predominantly of high quality there are relatively few areas of bad data, although it can readily be seen that the neighbour test is beginning to pick out the left hand side of the disc as a potential problem area, that is the area of the shadow.

Combined Weighting

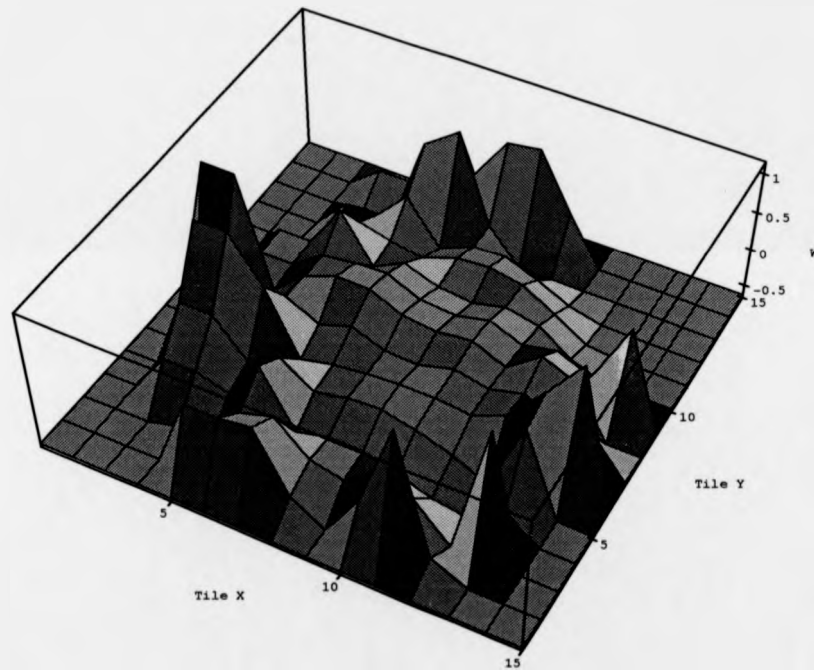


Figure 3.29: Combined Weighting Factors for Disc

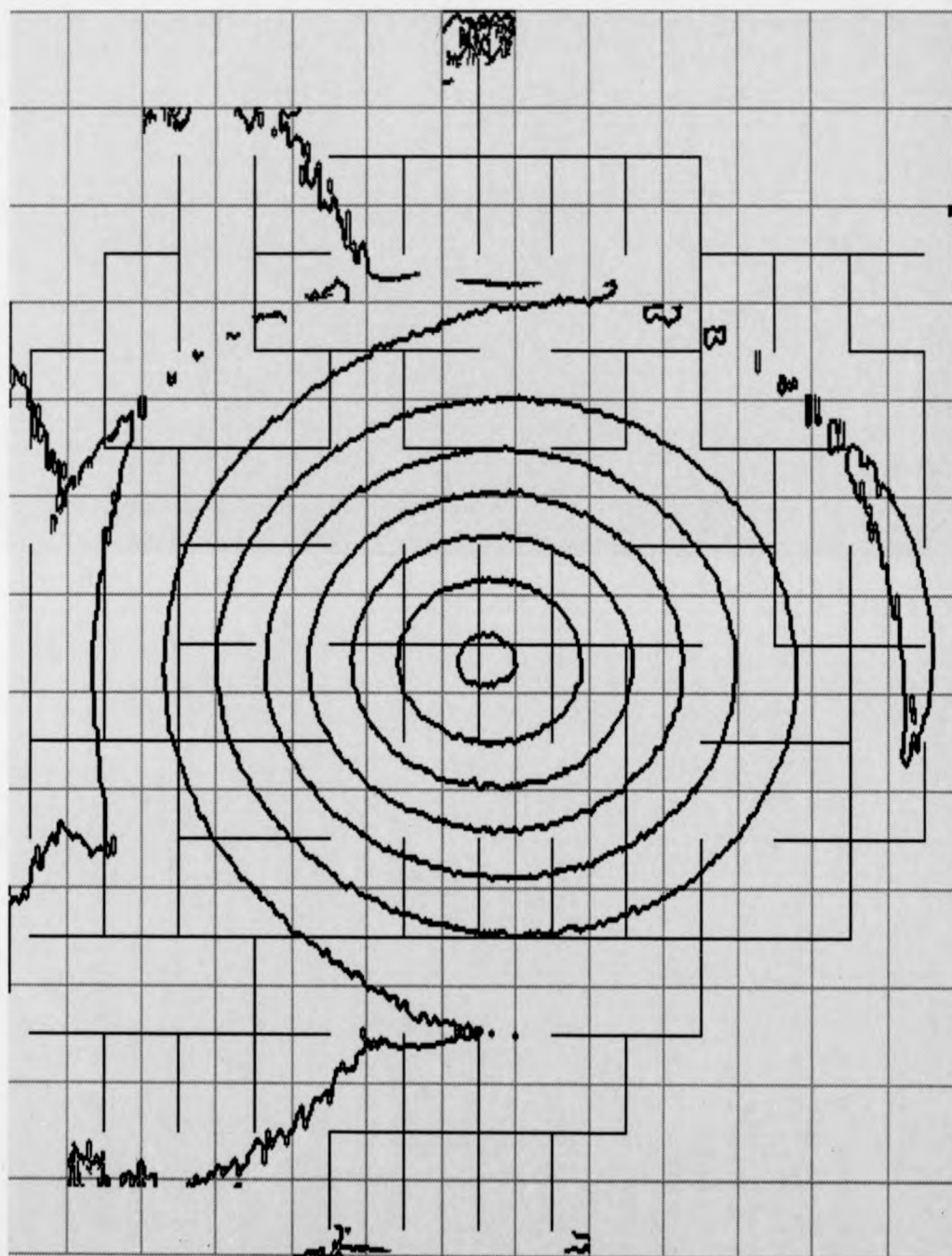


Figure 3.30: Edge Detection and Minimum Spanning Tile Tree for Disc

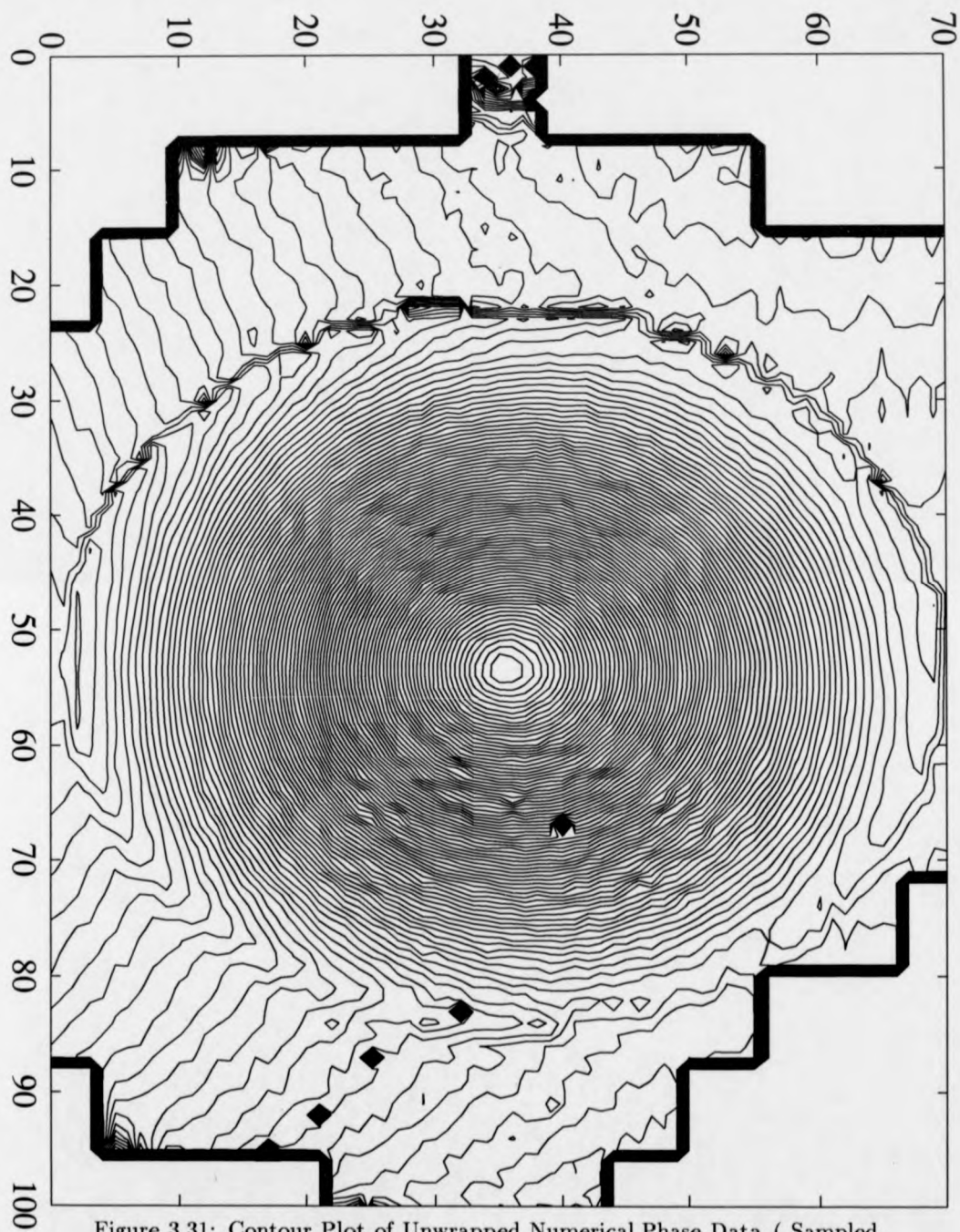


Figure 3.31: Contour Plot of Unwrapped Numerical Phase Data (Sampled at Every 5th Pixel in X) Showing Circumvention of Shadow Discontinuity

Figure 3.26 shows the result of tile by tile evaluation of low modulation noise. This plot does not show results for tiles which have been rejected. The Figure shows results for tiles which have less than 10 percent of their area as low modulation noise, any with a greater percentage are rejected (this 10 percent figure is definable by the operator). To see all of the low modulation points detected, refer to Figure 3.24. The low modulation test is better at detecting errors in a phase stepping system than an FFT one, as is evidenced by the phase stepping example later in this chapter. However, the test aids detection of aliased areas of the image, see top and bottom of Figure 3.24.

Figure 3.27 shows the result of a tile by tile analysis of fringe edge termination points. As can be seen this test has been very successful at identifying the shadowed side of the disc, and also an area of ragged fringe edges in the lower left corner of Figure 3.23. The fringe edges are shown more clearly by the edge detection of Figure 3.30. This is the best test of consistency for this particular image. Ragged fringe edges are more typical in images analysed by FFT methods than those analysed by phase stepping methods. Smearing is a feature of combining information from more than one pixel, as the Fourier transform requires. The phase stepping method computes phase in place, and so smearing is less of a problem.

Figure 3.28 shows the result of a tile by tile calculation of the number of points on fringe edges. As can be seen, the disc edge has again been picked out as an area of likely error. The values around the inside of the disc edge are low, as the fringes are more widely spaced here than near the centre of the disc. The basic assumption made by this test is that the less fringes present in a given area, the less likelihood there is of an error. There are filtering problems with FFT approaches, typified by the shadow discontinuity. In such cases this basic assumption is not always valid. The test breaks down in the region of the upper right hand corner of the disc (referencing Figure 3.23), here the wide spacing of fringes is due to the shadow. For the rest of the image, however, the test works well. The contributions of the other tests lessen the impact of this failing upon the final weighting, see Figure 3.29. Note, in Figures 3.25 and 3.27, how successful the neighbour agreement and the edge termination tests are in this area.

Figure 3.29 shows the combined (sum) of the weighting factors described above. This shows the weights that are assigned to the tile level graph's edges. It can be seen that the surround of the disc has been identified, generally, as a potential source of error, and that there is a particularly strong indication of likely error in the area of the shadow.

Figure 3.30 shows the path tree actually used to unwrap phase at the tile level, superimposed upon an edge detection of fringe edges. This tree is computed from the tile level graph, whose edge weights are, as mentioned above, shown in Figure 3.29.

Figure 3.31 shows a contour plot of the unwrapped numerical phase data. Noise points in this contour plot show as small black diamonds, bear in mind that only every 5th pixel on the x axis is plotted, and approximately every 3rd pixel in the y . The orientation of this plot is the same as that of Figure 3.23. As can be seen the shadow discontinuity has been contained. The rippling of the unwrapped data near the centre of the disc is due to the Fourier filter used to extract the side lobe. A full presentation of this data is given in Chapter 5, including a comparison of the deformation with that predicted by theory.

A mesh plot of the final deformation is shown in Figure 5.10. A side view of the unwrapped phase is shown in Figure 5.11. The same view is shown in Figure 5.12 after the phase values have been corrected for non uniformity of the carrier.

3.9.2 Phase Stepping Example of Tile Level Phase Unwrapping

The raw ESPI images, employed to generate this example, have been supplied by the Rover group research centre, Gaydon. One of the raw images employed is shown in Figure 3.32, it has been normalised to make it visible. The phase step between the input images was 90 degrees.

This example has been chosen to contrast with the previous case. The major difference is that the method, by which the wrapped phase map is computed, differs.

It is interesting to note the change which takes place, in the relative suc-



Figure 3.32: Raw ESPI Image

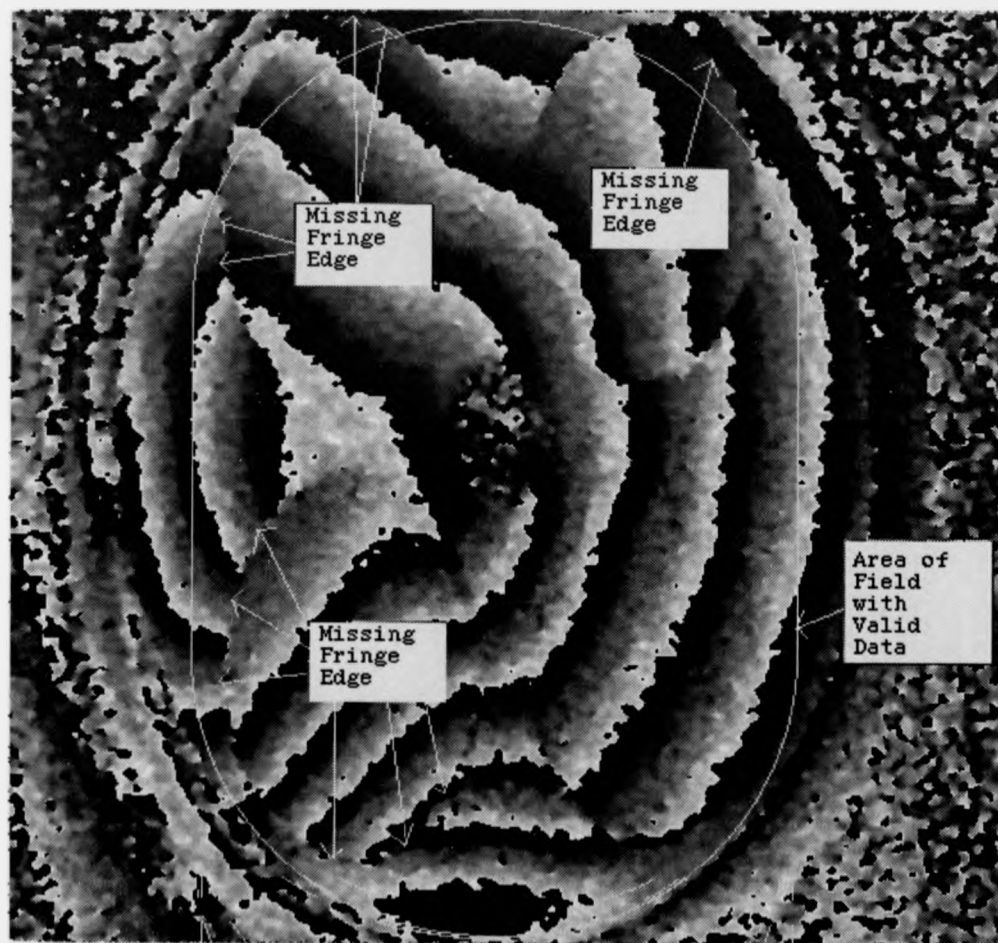


Figure 3.33: Wrapped Phase Map for Chamber Showing Problem Areas



Figure 3.34: Points of Low Modulation for Chamber (shown in white)

cessfulness of the various error detection mechanisms. The final weightings are computed on the relative significance of these error detection tests. The strategy therefore adapts to match the characteristics of the image.

The tile size has been selected as 64 pixels, including a 4 pixel overlap. This is comparable to the fringe spacing in the valid area of the image. It is larger than the size of the breaks between fringe edges. The reasons underlying the selection of tile size are described in Section 3.7.3.

Figure 3.33 shows the wrapped phase map for the chamber example. The wrapped phase map has been computed from three ESPI images. The ESPI images were spatially filtered by applying 4 iterations of a 3×3 averaging filter, prior to computation of phase.

The nature of the ESPI image is such that it consists of discrete points of information, captured in a video system. It is difficult to resolve a complex signal, because of the limited spatial resolution, especially in light of the filtering process which must be applied prior to phase calculation.

Figure 3.33 shows that there are a large number of discontinuities. As the signal increases in complexity, fringe edges are lost. The data around the central portion of the image is unambiguous, although there is a hole in its centre.

Figure 3.34 shows the points of low modulation which have been detected. As can be seen this test has effectively defined many of the problem areas in the image, including the central hole. Figure 3.36 shows the corresponding mesh plot of the tile by tile weighting factor for low modulation.

It should be noted that the orientation of the mesh plots with respect to the wrapped phase map is different between this series and the disc example. The orientation of the mesh plots with respect to the chamber image, shown in Figure 3.33, is such that the x axis of the mesh plot aligns with the horizontal of Figure 3.33 and the y axis aligns with the vertical. The origin for the mesh plots is in the bottom left hand corner of Figure 3.33.

Figure 3.35 shows a mesh plot of the tile neighbour agreement factor. As can be seen the hole at the centre of the image and the border of the frame have all been picked out as potential sources of error. There is a missing fringe edge near the bottom centre of the wrapped phase map and near the top left hand corner, labelled in Figure 3.33. Note particularly how the

neighbour agreement factor picks out these areas. These are not picked up in the low modulation test of Figure 3.36.

As mentioned above Figure 3.36 shows the weighting factor computed from the area density of low modulation points. For this analysis tiles were not rejected, even though they may have contained a large percentage of low modulation points. That is the area threshold was set at 100%. This was partly to show how the pixel phase unwrapping strategy circumvents low modulation pixels (the connection weights to such pixels are set to a notional infinity). If a lower threshold were employed, confidence would be greater in the remaining data, but large areas of the field would be lost.

Figure 3.37 shows the mesh plot for the edge termination factor. Refer to Figure 3.33. The centre and right hand sides of the frame have been found to have far fewer edge terminations than the left, top and bottom sides of the image. Areas of low modulation are not included in the count of fringe terminations. This explains why, for example, the central hole does not show in this plot.

Figure 3.38 shows the fringe density factor. This gives a similar result to the previous plot. It suggests that analysis should shy away from the top, bottom and left hand sides of the image and solve from the right hand side into the bottom right hand corner and then on around the centre. Again computation of this plot excludes points already identified as low modulation.

Figure 3.39 shows the combined (sum) of the above weighting factors. As can be seen the hole in the centre of the image has been clearly identified. The right hand side of the image is shown as less likely to contain errors than the left. The bottom left and top right hand corners of the image are shown to be particularly likely to contain errors.

Figure 3.40 shows the path tree actually used to unwrap phase at the tile level, superimposed upon an edge detection of fringe edges. This image also shows the areas of low modulation (in grey). This tree is computed from the tile level graph, whose edge weights are shown in Figure 3.39.

Figure 3.41 shows a contour plot of the unwrapped numerical phase data. Noise points in this contour plot show as small black diamonds, bear in mind that only every 5th pixel is plotted. Figure 3.42 shows the same data after post-processing with a 3×3 median filter to remove remaining noise spikes.

Figure 3.43 shows a grey scale image of the unwrapped phase, where black is low and white is high.

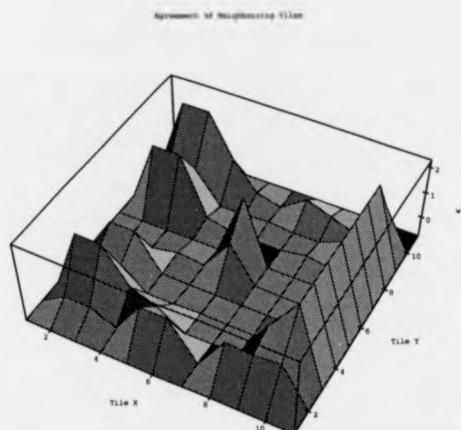


Figure 3.35: Weighting Factor for Chamber : Agreement between Neighbouring Tiles

3.10 Conclusion

It is noted that square tiles are not optimum for regional phase unwrapping. Octagonal tiles would be a better alternative, Figure 3.44 for example. These would permit consistency to be estimated across diagonals in addition to the compass points, north, south, east and west. The same reasoning applies to pixel level phase unwrapping. Note that the area of the tile overlap, in Figure 3.44, is the same in the eight directions.

This chapter has described the MSTT tiling technique for phase unwrapping, and given some examples. Further examples are given in Chapter 5.

The next chapter will discuss a variety of issues. These include; image capture using a CCD and frame store, image processing (spatial filtering and edge detection), and the application of parallel processing in fringe

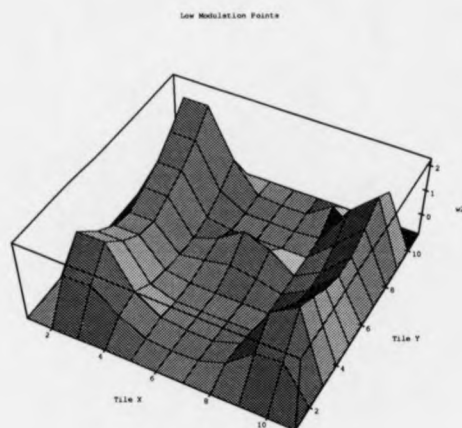


Figure 3.36: Weighting Factor for Chamber : Low Modulation Points

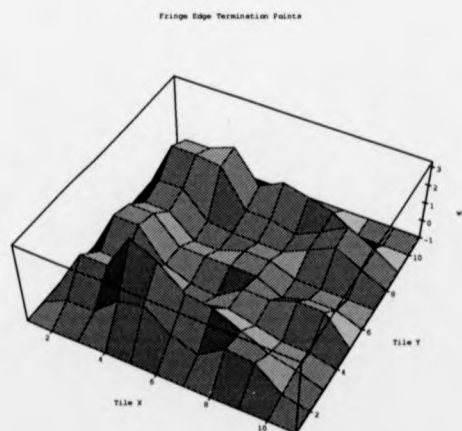


Figure 3.37: Weighting Factor for Chamber : Fringe Edge Termination Points

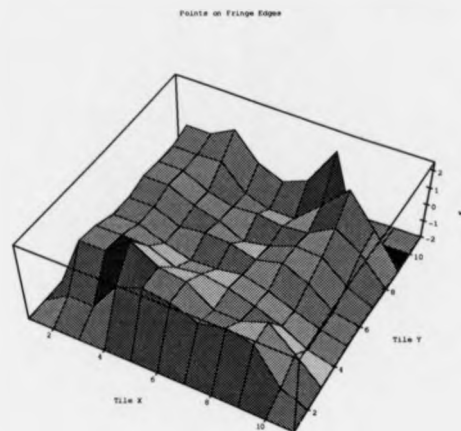


Figure 3.38: Weighting Factor for Chamber : Points on Fringe Edges

analysis. The parallel algorithm for Minimum Spanning Tree computation is introduced. It is shown that silicon devices optimised for parallel minimum spanning tree computation have been developed. These may one day find application in high speed fringe analysis systems.

Combined Weighting

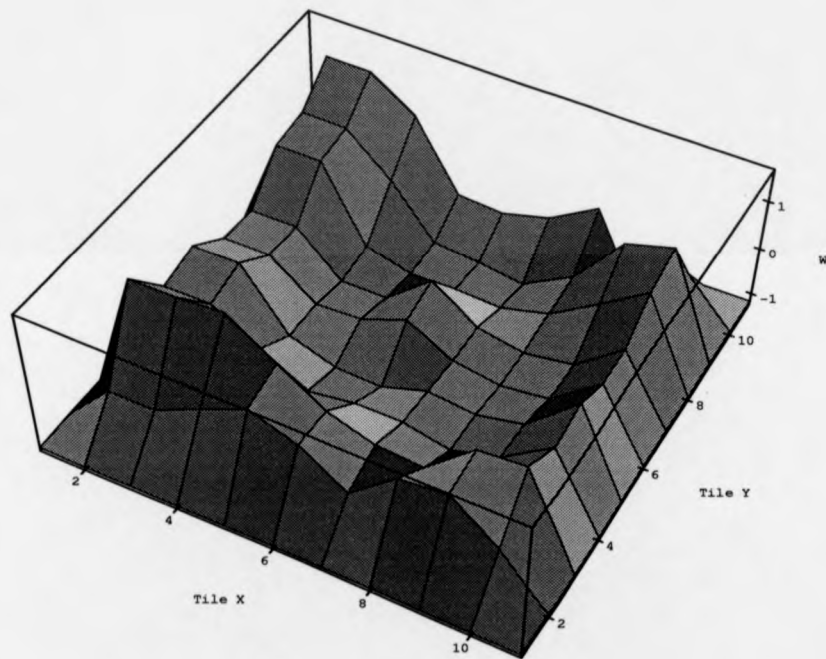


Figure 3.39: Combined Weighting Factors for Chamber

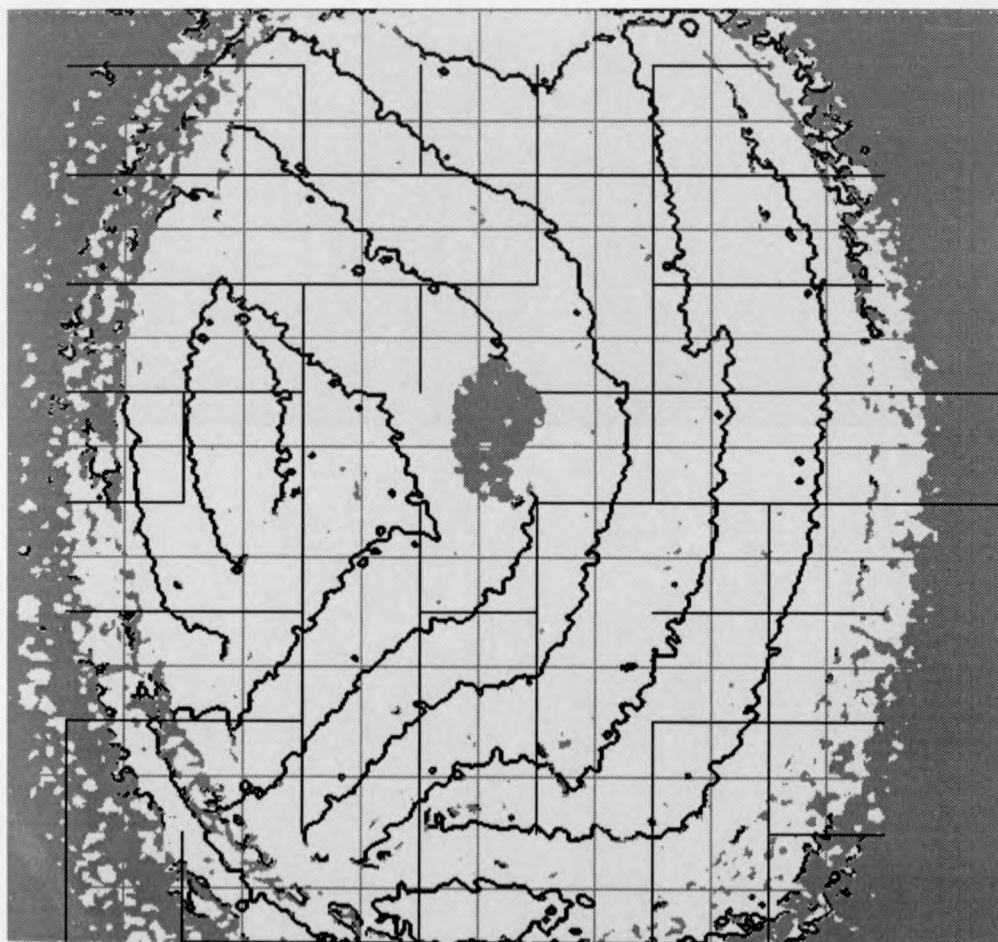


Figure 3.40: Edge Detection and Minimum Spanning Tile Tree for Chamber
(Includes Low Modulation Points in Grey)

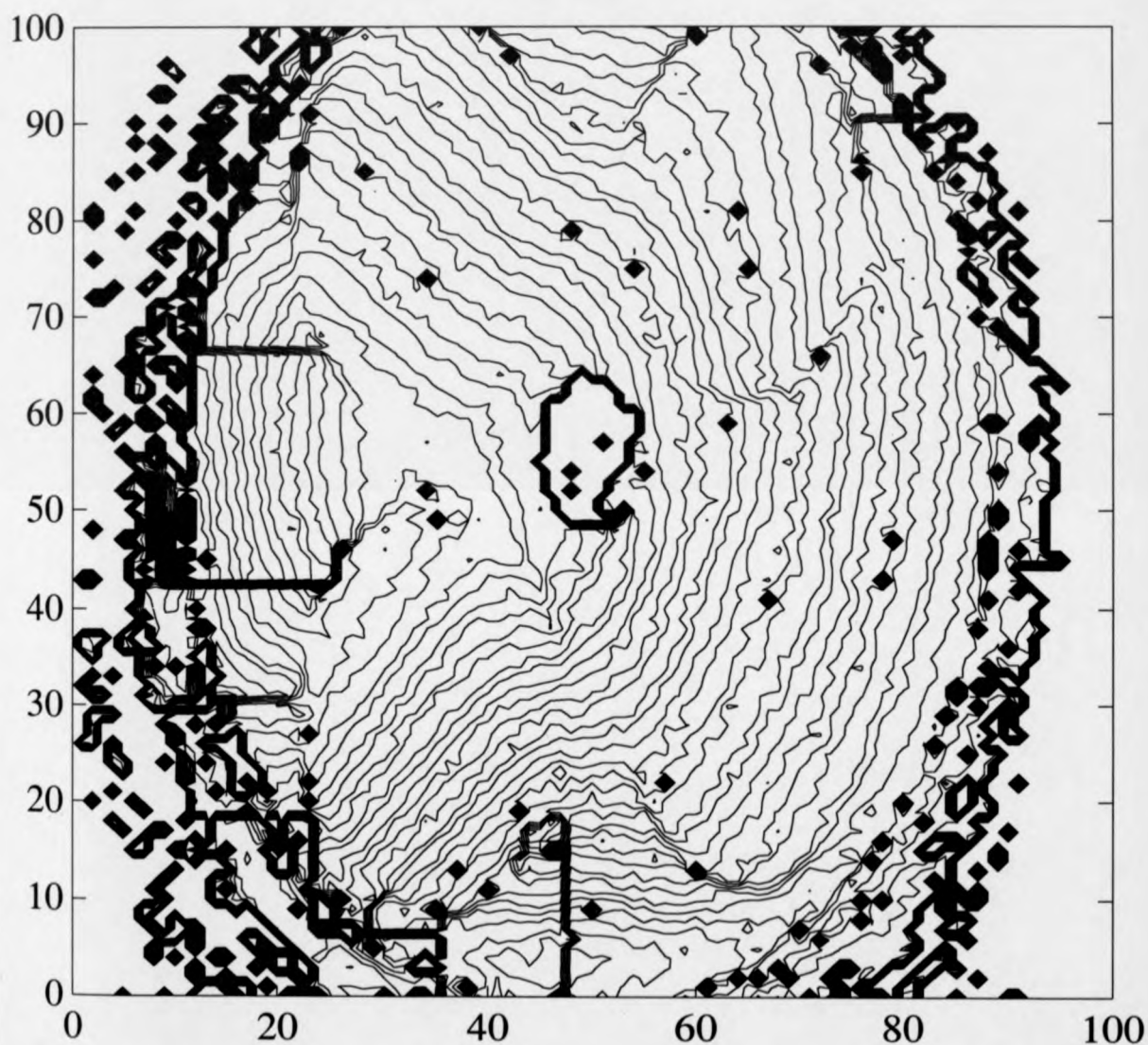


Figure 3.41: Contour Plot of Unwrapped Numerical Phase Data for Chamber
(Sampled at Every 5th Pixel) Showing Circumvention of Missed Fringe
Edges

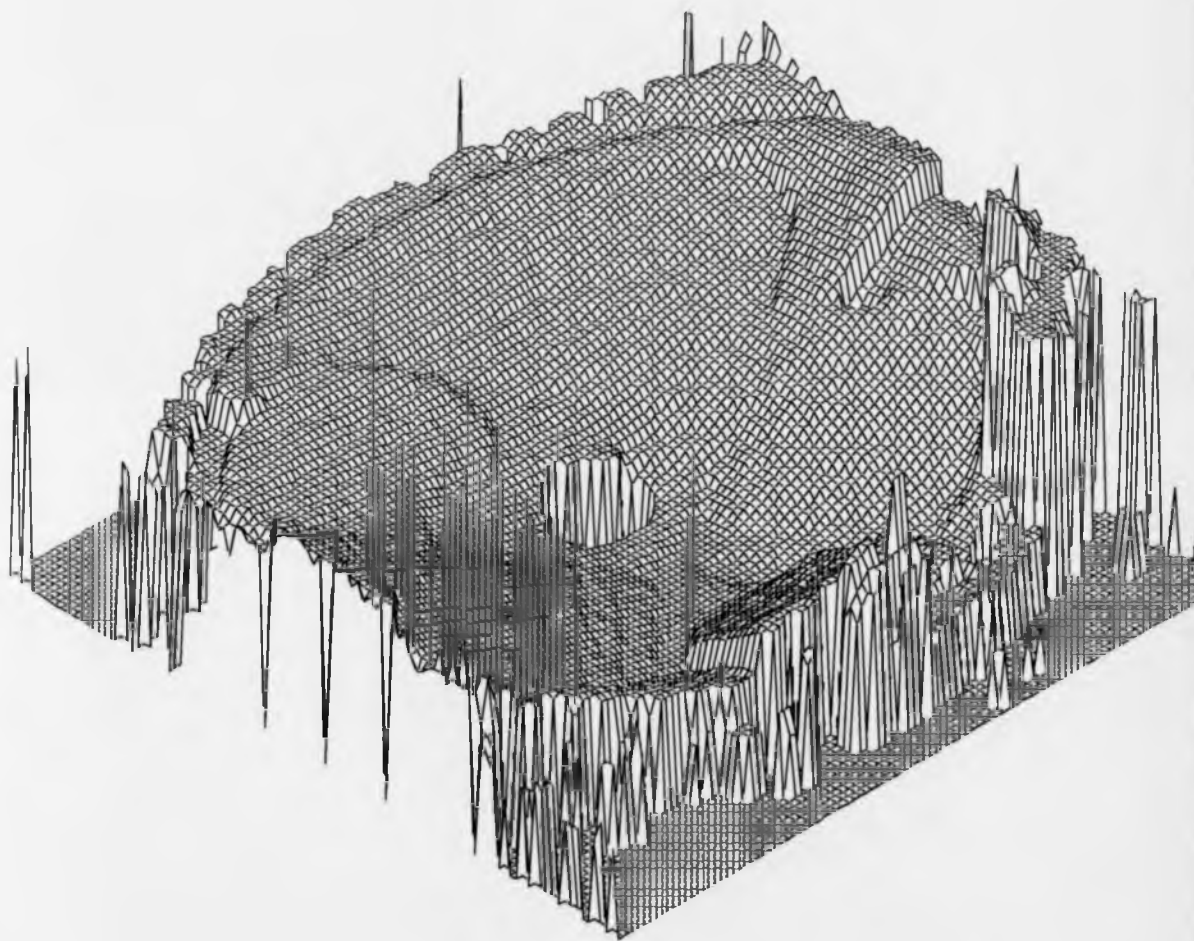


Figure 3.42: Mesh Plot of Unwrapped Numerical Phase Data for Chamber after Post Processing via 3 by 3 Median Filter (Sampled at Every 5th Pixel)

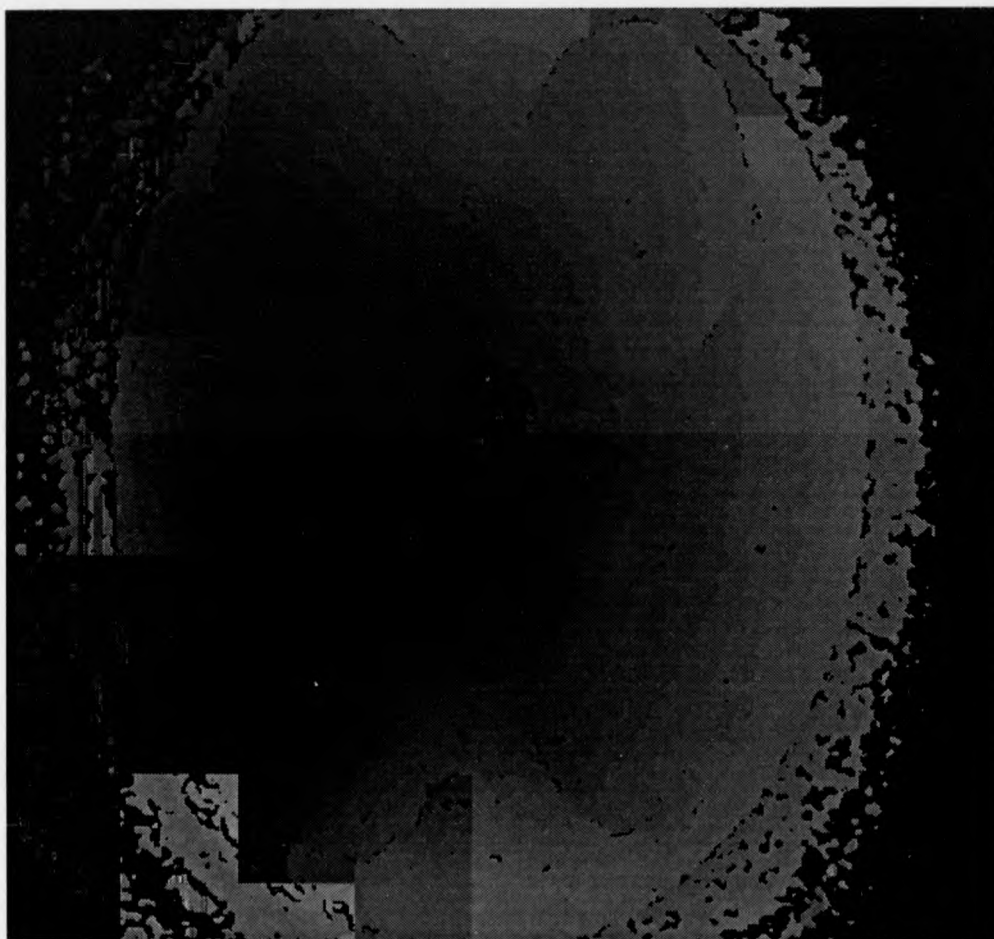


Figure 3.43: Grey Scale Image of Unwrapped Numerical Phase Data for Chamber Showing Circumvention of Missed Fringe Edges

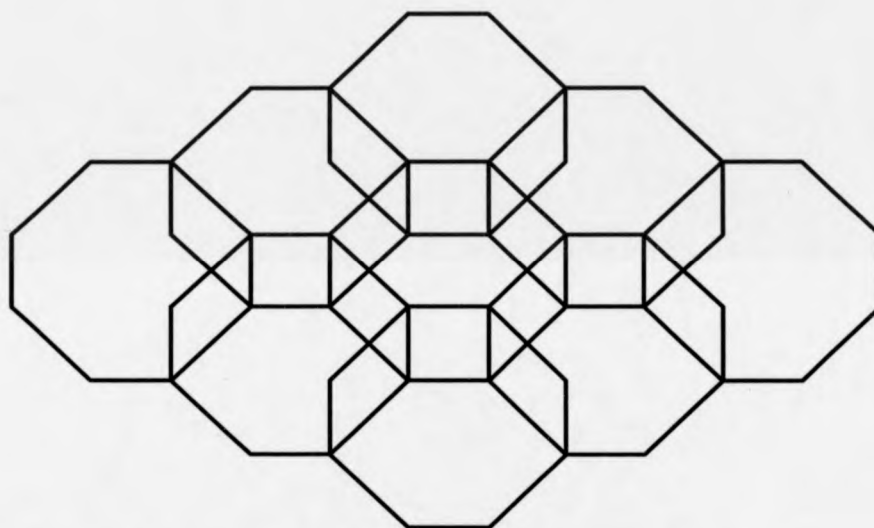


Figure 3.44: Octagonal tiles would allow better Consistency Testing (across diagonals)

Bibliography

- [1] A. Gibbons, *Algorithmic Graph Theory* , pp.40-41, Cambridge University Press, Cambridge, 1985.
- [2] N. Deo, *Graph Theory with Applications to Engineering and Computer Science* , pp.60-64, Prentice-Hall, London, 1974.
- [3] E. Shimon, *Graph Algorithms* , Pitman Publishing Limited, London, 1979.
- [4] J. J. Hopfield, D. W. Tank, "Computing with Neural Circuits: a Model", *Science* 233, pp. 625, 1986.
- [5] R. Thalman and R. Dandliker, "High Resolution Video-Processing For Holographic Interferometry Applied To Contouring And Measuring Deformations," *SPIE ECOOSA*, vol. 492, Amsterdam, 1984.
- [6] P. C. Donovan, D. R. Burton, M. J. Lalor, "Fourier Analysis of Partial Field Fringe Patterns", *Applied Optics Digest*, 17th-20th September, pp.325-326, 1990.
- [7] J. Davies, C. Buckberry, "Applications of a Fibre Optic TV Holography System to the Study of Large Automotive Structures", *SPIE*, 1162, 1989.
- [8] R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Tech. Journal*, vol. 36, pp.1389-1401, Nov. 1957.
- [9] H. C. Andrews, *Introduction to Mathematical Techniques in Pattern Recognition* , John Wiley & Sons, Inc., ISBN 0-471-03172-0, 1972.
- [10] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Vibration Measurements Using Dual Reference Beam Holography", *SPIE proceedings*

1084, Stress and Vibration: Recent Developments, March, pp. 218-239, 1989.

- [11] C. Buckberry, J. Davies, "Digital Phase-Shifting Interferometry and its Application to Automotive Structures", Applied Optics Digest, 17th-20th September, pp. 275-276, 1990.

Chapter 4

Image Capture and Processing

4.1 The CCD Camera

Recently the CCD (Charge Coupled Device) camera has gained particular prominence as a tool for the capture of interferograms. This device contains a regular grid of sensing elements which eliminates some of the geometrical distortions of earlier imaging systems, using the Vidicon tube. The sensitivity of the CCD device can be very great, it has become an important device in astronomy for looking at faint objects such as distant stars.

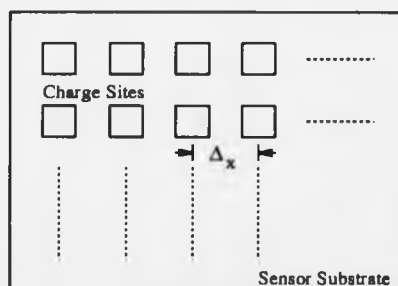


Figure 4.1: CCD Cells Arranged on Rectangular Grid

CCD technology involves a discrete number of charge storage sites and a charge transfer mechanism. The array is usually organised as a rectangular grid, see Figure 4.1.

The basic CCD cell is an enhancement mode MOS device that behaves like a capacitor, see Figure 4.2. The device is formed by diffusing an impurity,

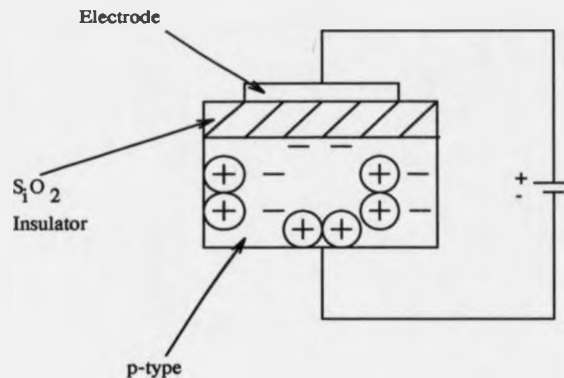


Figure 4.2: 1D Slice for p-type CCD Cell

either a group III or V element, into pure silicon (a group IV element). This creates a semiconductor device upon which an insulator (typically, silicon dioxide) is formed. Finally, electrodes are deposited on top of the insulator to facilitate charge transfer. The minority carriers (holes or positive charge regions in n -type semiconductors and electrons in p -type semiconductors) are then moved by applying suitable voltages to the transfer conductors. In Figure 4.2 a p -type semiconductor is used for illustration. The charge of interest (negatively charged electrons) is attracted to the electrodes when a positive voltage is applied to them (with respect to the substrate). This creates charge packets stored in potential wells under the electrode. [1]

To read out the image, the collected charge is moved from cell to cell and eventually out of the array. Figure 4.3, illustrates how three electrodes together make up a complete cell, the positive potential being applied only to every third electrode. The read out is achieved by sequencing a change in the cell electrode voltages. There are of course many variations on this basic theme.

There is a wide variety of CCD devices available, many designed with specific applications in mind. Some for example are sensitive to specific light wavelengths, others have been incorporated with a phosphor which glows when struck by X-rays.

At the present time most image analysis systems digitise the intensity of

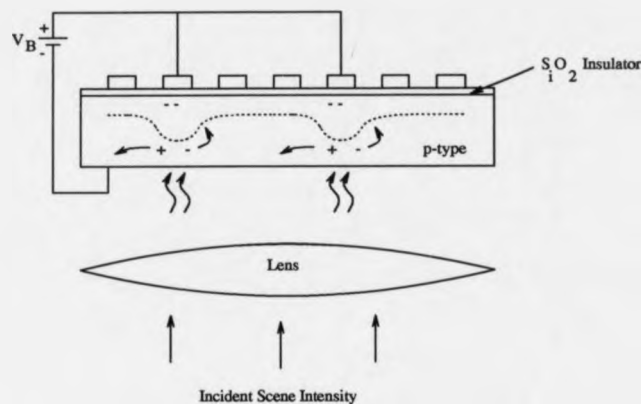


Figure 4.3: Generation of Minority Carrier Charge Packet from Incident Illumination (1-D Slice)

a standard CCD camera to 8-bits of intensity resolution. The standard CCD camera has a spatial resolution of the order of 500 by 500 pixels.

Devices are becoming available with significantly greater resolutions than this, both in terms of intensity and spatial resolutions. For example a liquid nitrogen-cooled CCD array (Thomson, CSF) is being employed in systems which digitise to 14 bits of accuracy. An application of this device is described in reference [2]. The digitisation produces a range of values from 0 to $2^{14} - 1$ (16383). The CCD has an array of 384×576 pixels having dimensions of $23 \times 23 \mu\text{m}$. Liquid nitrogen cooling of the array results in a substantial reduction of the thermal background signal associated with the array when it is not exposed to light. The average back ground level, in the referenced application, was found to be 300. This background was found to be nearly uniformly distributed across the array. The usable dynamic range has been claimed to exceed 10^3 . Another application employing a 14-bit intensity resolution CCD camera is given in [3].

In terms of spatial resolution Kodak market a CCD camera with a spatial resolution of 4096×4096 pixels. The read out time of the device is of the order of several seconds. It requires a dedicated frame store.

4.2 Capturing the Image of a Scene

An image is obtained by sampling the intensity of light striking the face of the discrete sensor array, after passage through some optical elements to provide focus etc. This immediately leads to the issue of discrete sampling of an inherently continuous function, that is the light coming from the scene.

For simplicity sampling is considered for a one dimensional array of sensing elements. Suppose the light from the scene, a continuous signal, is represented as $h(x)$, where x is a distance measured across the sensor array. Let Δ_x denote the spacing of the sensor elements, and therefore the spacing of intensity samples. The sequence of intensity samples across the array is then

$$h_n = h(n\Delta_x) \quad n = 0, 1, 2, \dots, N-1 \quad (4.1)$$

For any sampling interval Δ_x , there is also a special frequency, a spatial frequency in this case, f_c , called the Nyquist critical frequency, given by;

$$f_c \equiv \frac{1}{2\Delta_x} \quad (4.2)$$

If an intensity sine wave of the Nyquist critical frequency is sampled at its positive peak value, the next adjacent sensor will capture its negative trough. The next sensor in the chain will capture the next peak, and so on. Expressed otherwise: Critical sampling of a sine wave is achieved with two sample points per cycle.

If a continuous function $h(x)$, sampled at an interval Δ_x , happens to be band-width limited to frequencies smaller in magnitude than f_c , then the function $h(x)$ is completely determined by its samples h_n .

The effect of sampling a continuous signal that is not band-width limited can be serious. In this case, all of the power spectral density, which lies outside of the frequency range $-f_c < f < f_c$ is forced erroneously into this range. This phenomenon is called aliasing. Any frequency component outside of the frequency range $(-f_c, f_c)$ is aliased, that is falsely translated into that range by the very act of discrete sampling. [15]

4.2.1 The Effect of Intensity Quantization

The above analysis and the Nyquist sampling criterion assume that at each sampling position the exact value of intensity is recorded. In a digital system with a finite register length this is plainly not the case. In almost all systems in operation today, intensity is quantized into samples with an 8-bit register length. The camera itself does not usually perform this digitisation process itself. This is most often accomplished by a frame capture board which receives an analogue video signal from the camera.

In the analysis of quantization, the discussion will follow a single sensor and its sampling, in time, of an analogue intensity signal. The previous section considered spatial sampling of an image *across* an array of such sensors, *not* temporal sampling. The distinction is emphasised to avoid confusion.

The following discussion is derived from reference [11]. Imagine that the register being used to record intensity has a length of b bits. The register is to record a binary measurement between 0 and $(1 - 2^{-b})$, in steps of (2^{-b}) . Suppose that b_1 bits would be required to 'exactly' specify the intensity measurement, and b is the number of bits to which the measure is truncated. In this case $b < b_1$. The effect of truncation is to discard the least significant $(b_1 - b)$ bits, and consequently the magnitude of the number after truncation is less than or equal to the magnitude before truncation.

If the number before truncation is denoted by a , and after truncation by $Q\{a\}$, then the truncation error is;

$$E_T = Q\{a\} - a \quad (4.3)$$

The largest error occurs when all the bits discarded are unity, in which case truncation reduces the value of the register by $(2^{-b} - 2^{-b_1})$, thus;

$$-(2^{-b} - 2^{-b_1}) \leq E_T \leq 0 \quad (4.4)$$

Generally, it is reasonable to assume that 2^{-b_1} is very much less than 2^{-b} , and so;

$$-2^{-b} < E_T \leq 0 \quad (4.5)$$

In the digitisation process it is assumed that the input samples h_n , which are samples in time at a single sensor, are truncated to the next lowest quantization level, to obtain the quantized samples \bar{h}_n . In order to ensure that the unquantized samples are within the range of the b -bit number, it must also be assumed that the intensity waveform is normalised, so that;

$$0 \leq h_n < 1 \quad (4.6)$$

If the exact value of an input signal falls outside of the range indicated, then additional distortion results. The quantized value $1 - 2^{-b}$ is assigned to all samples equal to or greater than 1. This clipping of the input intensity is highly undesirable, and it must be eliminated by reducing the amplitude of the input until Equation 4.6 is satisfied.

In the frame capture application, this is usually accomplished by reducing the aperture of the camera lens (increasing the f-stop number). In the system employed, with b as 8 bits, the output look-up-table of the frame store was configured so that the maximum in range value ($1 - 2^b$) displayed as red, the rest of the scale being shown as a scale of grey. In this way a real time indication of the probability of clipping was obtained simply by observing the amount of red in the captured image. Two further controls of the digitisation process were available. The gain and offset of the frame capture board's analogue to digital converter could be varied, see Figure 4.4.

The offset register could be used to adjust a DC voltage, applied to the input analogue signal, in order to clamp the black level to 5 volts so that the darkest part of the image would be digitised as binary 0. The gain register could be used to adjust the white level to 0 volts, so that the brightest part of the image would be digitised as the maximum permitted value. See reference [12], for the hardware description. As an additional aid, a histogram of the image on display could be computed to explore the spread of the signal over the digitisation range.

Figure 4.5 shows equivalent representations of the quantization process. That is;

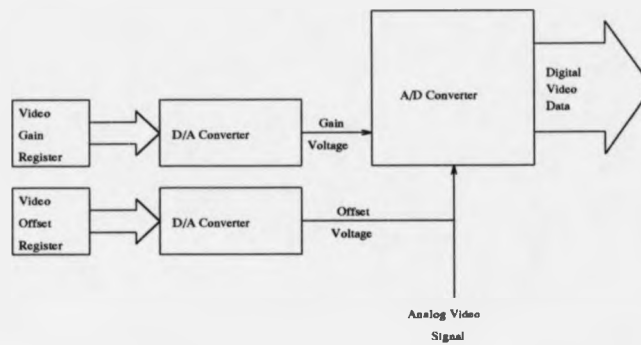


Figure 4.4: Input Video Signal Gain and Offset Control

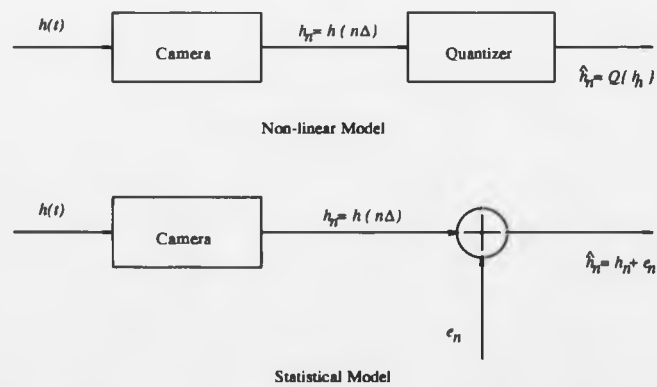


Figure 4.5: Representation of Sampling the Analog Intensity Signal

$$\hat{h}_n = Q\{h_n\} = h_n + e_n \quad (4.7)$$

where h_n is the exact sample and e_n is called the quantization error. Since truncation was assumed;

$$-2^{-b} < e_n \leq 0 \quad (4.8)$$

e_n is unknown. A statistical model, as shown in Figure 4.5 has been used to represent the effects of quantization in sampling. In particular, the following assumptions have been made;

- i) The sequence of error samples e_n is a sample sequence of a stationary random process.
- ii) The error sequence is uncorrelated with the sequence of exact samples h_n .
- iii) The random variables of the error process are uncorrelated; i.e., the error is a white-noise process.
- iv) The probability distribution of the error process is uniform over the range of quantization error.

[11]

In a heuristic sense, the assumptions of the statistical model appear to be valid if the signal is sufficiently complex and the quantization steps are sufficiently small so that the amplitude of the signal is likely to traverse many quantization steps in going from sample to sample [11].

A noise level in the analogue part of the capture system has caused quantized intensity measures to traverse several quantization levels within a series of temporal samples. This noise is either generated within the CCD array or possibly in the carriage of this signal to the digitiser. This, paradoxically, may be used as an aid to the digitisation process. The distribution(s) of the noise have been explored. There is a distribution to the samples, and so, a large number of frames might be combined to extract a mean estimate for the intensity at each pixel.

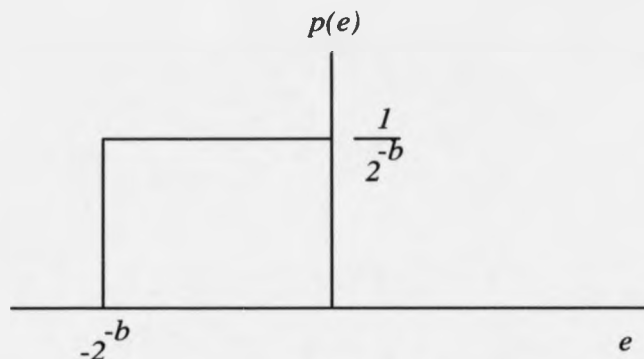


Figure 4.6: Probability Density Function of Quantization Error with Truncation

Truncation obviously contributes to this distribution. It is assumed that the probability distribution of the error is as shown in Figure 4.6. It is assumed that the error is independent of the signal.

The mean of the quantization noise, in this case, may be seen from Figure 4.6;

$$m_e = -\frac{2^{-b}}{2} \quad (4.9)$$

the variance may be computed as;

$$\sigma_e^2 = \frac{2^{-2b}}{12} \quad (4.10)$$

For the arrangement employed, with an 8 bit digitiser, the mean and variance of the quantisation noise are given by;

$$m_e = -0.5 \quad (4.11)$$

and

$$\sigma_e = 0.2875 \quad (4.12)$$

That is, accepting the assumptions indicated above and considering digitised values to lie in the range 0 to $(256 - 1)$, rather than 0 to $(1 - 2^{-b})$. The

mean value of the truncation noise is particularly relevant in describing the noise distribution of the camera/digitiser pair, as will be seen in the following section.

4.3 An Experiment to Investigate the Distribution of Noise in a Particular Camera/Digitiser Pair

This experiment has been devised to investigate the noise level of a particular CCD camera/digitiser combination.

The experiment employs a Helium-Neon laser as a light source. Ordinary room lighting is inappropriate as this fluctuates with the frequency of mains electricity.

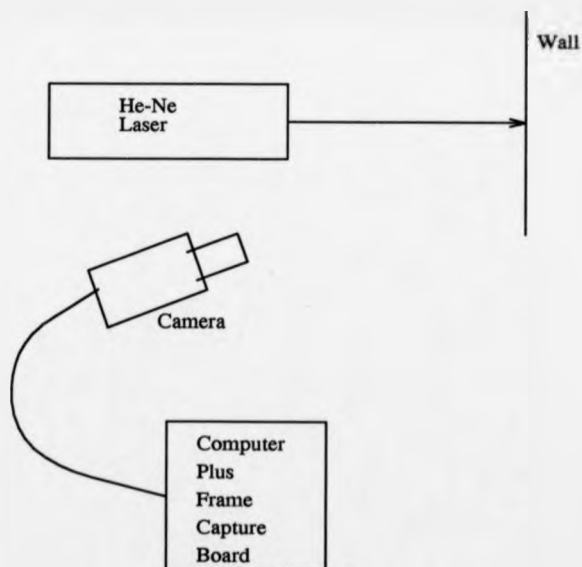


Figure 4.7: Experimental Arrangement for Camera Noise Experiment

The laser is arranged so as to project a patch of illumination upon a stable wall. See Figure 4.7. The camera is arranged so as to view the illuminated patch.

The system is left to become stable. This precaution is taken as otherwise motion of the air or the object (on to which the light is projected) causes low frequency fluctuations in intensity measurements.

Data are collected at a number of pixel positions over a sequence of 1000 video frames. In the particular system employed, the hardware required $\frac{1}{25}$ th of a second to capture a video frame. The software required a short interval to read the pixel values, and to request the hardware to collect another frame. However, because the frame capture board required time to resynchronise with the CCD camera (the capture board relied on the camera for its timing), only every second frame generated by the video camera was collected. Therefore, to capture 1000 frames, required 1 minute and 20 seconds (that is $(\frac{2}{25}) * 1000$ seconds). The process was repeated for 3 different values of video gain to explore the relationship between gain and noise.

The χ^2 goodness of fit test has been employed to compare the sampled data against the normal distribution. Suppose that the noise distribution is X , which is a discrete random variable. Further suppose that X_1, X_2, \dots, X_N represents a sample of X , recorded as described above, of size $N = 1000$. The null hypothesis is

$$H_0 : P(X \leq x) = \Phi(x; \mu, \sigma) \quad (4.13)$$

the alternative is

$$H_1 : P(X \leq x) = F(x) \neq \Phi(x; \mu, \sigma) \quad (4.14)$$

The parameters μ and σ are unknown, and are therefore estimated from \bar{X} and s_{n-1} respectively. The samples are quantised and so the range of values they may take is limited. The sample data is divided into r parts, where $r = (\max(X) - \min(X) + 1)$. The (y_i) , $(i = 1, 2, \dots, r)$ are used to denote the possible values of X . The o_i are used to represent the number of sample values within a given interval (y_{i-1}, y_i) . That is, the (o_i) , $(i = 1, 2, \dots, r)$ are observed frequencies. The expected frequencies e_i , for $(i = 1, 2, \dots, r)$, as given by the normal distribution are computed from

$$e_i = N \cdot \left(\Phi\left(\frac{y_i - \bar{X} - m_e}{s_{N-1}}\right) - \Phi\left(\frac{y_{i-1} - \bar{X} - m_e}{s_{N-1}}\right) \right) \quad (4.15)$$

Note the use of m_e , the mean truncation error. This factor biases the mean of the sampled data so that the expected values of frequency, computed from the normal distribution, are aligned with the sampled data. That is, an estimate of the true mean of the analogue signal is obtained by correcting the mean of the sampled data by the mean of the truncation noise.

Digitised Value	Observed Frequency (O)	Expected Frequency (E)	$\chi^2 = \frac{(O-E)^2}{E}$
70- >71	5	10.2275	2.6719
72	31	30.8894	0.0004
73	66	81.2994	2.8791
74	152	155.3998	0.0744
75	240	215.7638	2.7224
76	270	217.6272	12.6037
77	134	159.4614	4.0654
78	75	84.8723	1.1483
79	18	32.8071	6.6830
80- >81	9	11.6521	0.6036
Total	1000	1000.0000	33.4522

Table 4.1: Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 1

Tables 4.1, 4.2, 4.3 and the corresponding Figures 4.8, 4.9, 4.10 show a typical set of results, taken at different and increasing values of gain. Table 4.3 shows a summary of the results of the χ^2 tests for the data of Figures 4.9, 4.10 and 4.10. The data have been subdivided as shown in the bar charts, with one bin for each digitised intensity level present in the sample data. The number of degrees of freedom is equal to the number of bins r , minus 1, minus another 1 for each of the estimated parameters of the normal distribution μ and σ . The number of degrees of freedom is usually $r - 3$. However, if a bin contains less than 5 samples, it must be combined with a larger bin, and each time this occurs another degree of freedom is lost.

Noise is known to be present in the signal before digitisation as its spread varies with the gain of the analogue- to-digital converter, as reflected in the

Digitised Value	Observed Frequency (O)	Expected Frequency (E)	χ^2 $= \frac{(O-E)^2}{E}$
88- >89	18	30.7896	5.3126
90	49	54.8029	0.6144
91	101	103.7593	0.0734
92	170	156.5154	1.1618
93	205	188.1100	1.5165
94	192	180.1357	0.7814
95	141	137.4417	0.0921
96	73	83.5519	1.3326
97	35	40.4661	0.7384
98	11	15.6132	1.3631
99- >100	5	8.8142	1.6505
Total	1000	1000.0000	14.6368

Table 4.2: Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 2

Digitised Value	Observed Frequency (O)	Expected Frequency (E)	χ^2 $= \frac{(O-E)^2}{E}$
113- >114	10	9.8035	0.0039
115	8	18.5439	5.9952
116	39	39.7396	0.0138
117	66	72.2359	0.5383
118	120	111.3785	0.6674
119	144	145.6714	0.0192
120	193	161.6132	6.0956
121	160	152.0935	0.4110
122	123	121.4156	0.0207
123	65	82.2177	3.6056
124	41	47.2254	0.8206
125	23	23.0090	0.0000
126- >127	6	15.0528	5.4444
Total	1000	1000.0000	23.6357

Table 4.3: Goodness of Fit Test. That is, a Comparison of the Noise Distribution against the Normal Distribution, for Gain 3

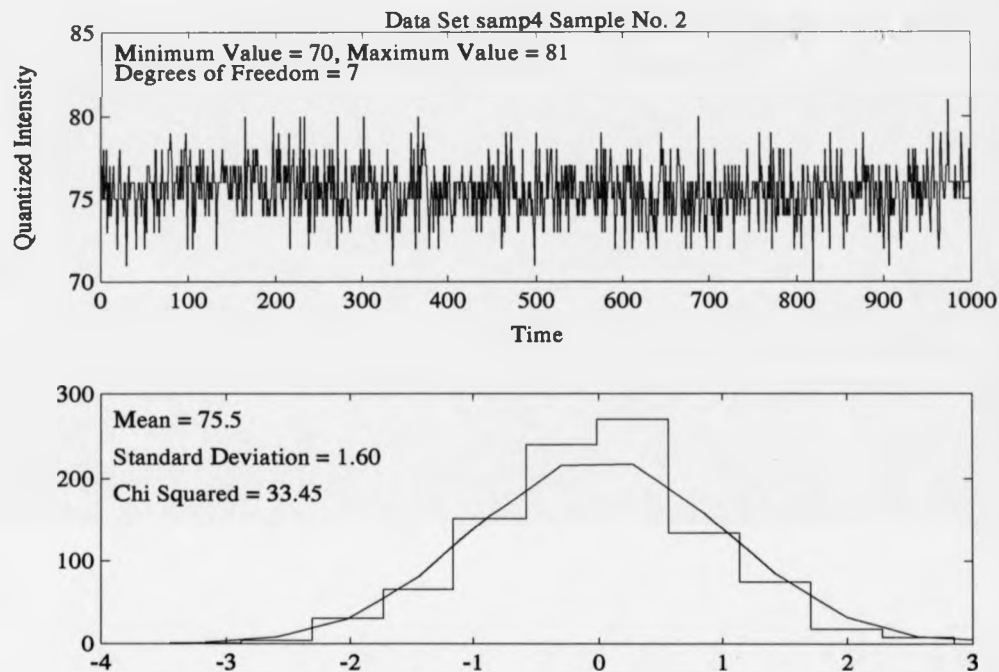


Figure 4.8: Result of Camera/Digitiser Noise Experiment, Gain 1

\bar{x}	s_{n-1}	χ^2_c	Degrees of Freedom	Result of Test
75.5	1.60	33.45	$12 - 5 = 7$	$P(\chi^2 \leq 33.45) \approx 100\%$
93.3	1.92	14.64	$13 - 5 = 8$	$P(\chi^2 \leq 14.64) = 93.0\%$
120.1	2.28	23.64	$15 - 5 = 10$	$P(\chi^2 \leq 23.64) = 99.1\%$

Table 4.4: Summary of Goodness of Fit Tests, Comparing the Noise Distribution with the Normal Distribution

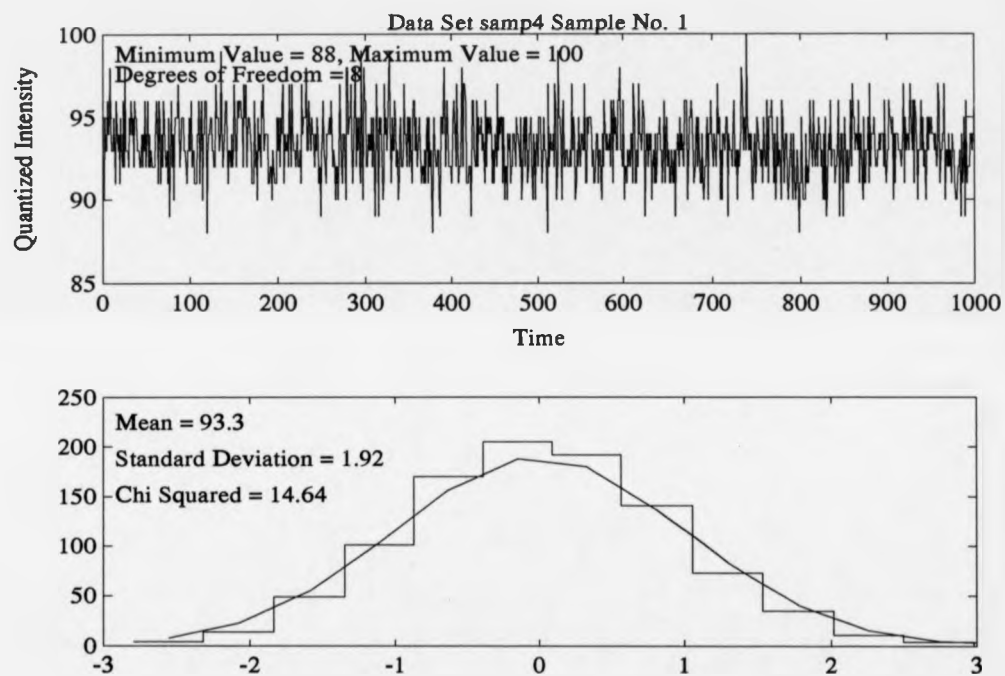


Figure 4.9: Result of Camera/Digitiser Noise Experiment, Gain 2

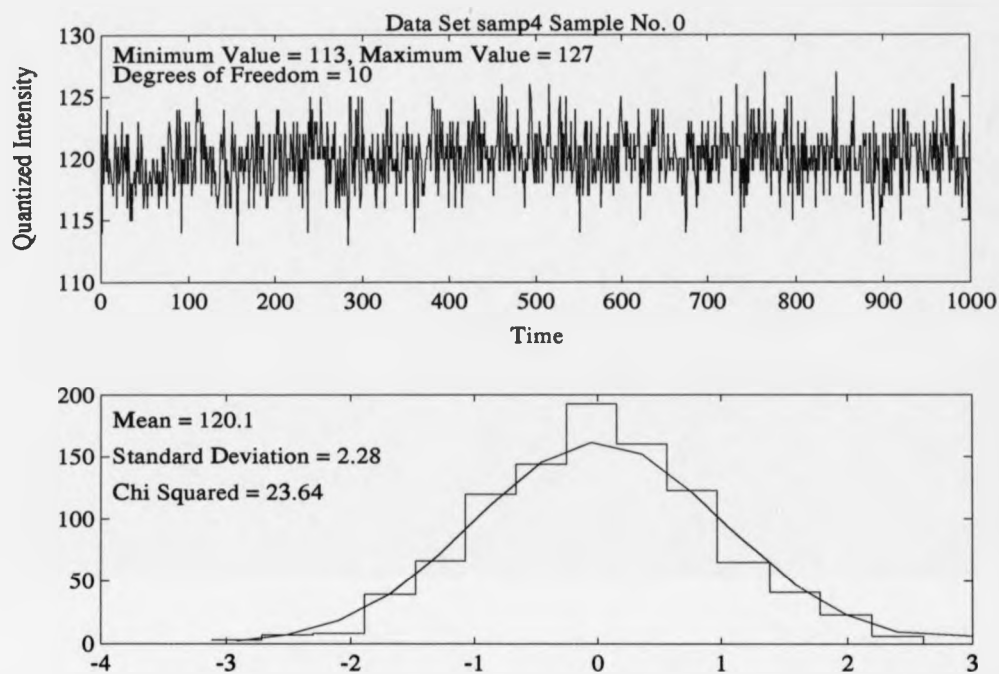


Figure 4.10: Result of Camera/Digitiser Noise Experiment, Gain 3

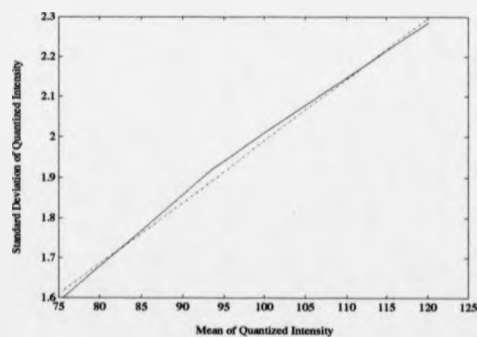


Figure 4.11: Standard Deviation of Intensity Against Mean for Varying Gain

mean of the digitised intensity, see Figure 4.11. Quantization reduces the mean as explained above.

The noise level varies with the digitiser gain, as does the digitised signal. The maximum deviation from the mean intensity, expressed as a percentage of the mean, varied as 7.9 % for gain 1, 6.4% for gain 2 and 5.8 % for gain 3.

The results indicate, see Table 4.3, that the quantized noise distribution, with contributing effects from both the analogue domain and the digitisation process, is on the bounds of possibility of being normally distributed at certain values of gain, but is far from it at others. For sample 2, of Table 4.2 and Figure 4.9, the χ^2 test gave a 7% or 1 in 14 chance that the sample originated from a normal distribution. For sample 3, Table 4.3 and Figure 4.10, it gave a 0.9% chance, roughly 1 in 100, that the original distribution was normal. For sample 1, Table 4.1 and Figure 4.8, it gave a negligible chance of the distribution being normal. This last sample is the one with the lowest gain and mean intensity, it deviates most from the normal in the centre. It should be remembered that the intensity signal actually arriving at the analogue-to-digital converter of the digitiser is very similar for the three samples, as sampling has been performed at the same pixel in the same scene. It seems, then, that the gain of the converter effects the distribution of noise in the digitised signal. This suggests a non-linear gain. As such factors as gain are changed often, perhaps from shot to shot, it is important to calibrate the camera and assess the affects of the settings employed.

4.3.1 Effect of the Camera Response Upon the Accuracy of Fringe Analysis Techniques

There is a fundamental difference between the phase stepping approach to fringe analysis and the FT method in terms of the factors which determine accuracy. The accuracy of measurement in the phase stepping system is dependent upon intensity. Whereas in the FT method it is dependent upon the accuracy with which the fringe positions are spatially determined. In relation to the preceding discussion, a noise level on the CCD detector is seen to be more damaging in a phase stepping system than in an FT based system. In addition, non-linearity of the CCD detector in its response to

light intensity is also more damaging in a phase stepping system.

4.3.2 Image Data Format

There is an abundance of data formats for recording images. These include GIF, IFF, PCX, PGM, TIFF etc. The main advantage of adopting a standard format is ease of data transfer from one system or software package to the next. In the Interferometric application it was of paramount importance that any format adopted should have sufficient capacity to encode intensity values accurately. It was also desirable to have the latent capacity to cope with higher resolution capture devices, both in terms of intensity resolution and image size. Practically speaking this meant that formats with an 8 bit maximum range for pixel values were not acceptable. The TIFF format has been adopted [13].

This format permits an arbitrary number of bits per pixel resolution, up to 32 bits. It is a tag based file format that is designed to promote the interchange of digital image data.

The standard has been developed in order to take advantage of the varying capabilities of scanners, frame stores and similar devices. TIFF is therefore designed to be a superset of existing image file formats. A high priority has been given to structuring the data in such a way as to minimize problems in updating the format. The standard is intended to be independent of specific operating systems, filing systems, compilers, and processors.

[13]

Individual fields are identified with a unique tag. This allows particular fields to be present or absent from the file as required by the application. The standard supports some very useful fields, such as ImageDescription. This, for example, allows a note to be kept, of the subject of an image, and also the image processing functions which have been applied to it.

4.4 Spatial Smoothing

Spatial smoothing filters are typically used for noise removal. They have been extensively applied to filtering noise from interference patterns [14]. Their

properties are worthy of review, in order to understand the distorting effects they have upon the image data.

The crucial point about spatial filters is that once a filter with a given window size has been applied, the spatial resolution of the result is reduced to the size of the window employed. That is, suppose for example a 3×3 pixel window is applied, the resulting value of the central pixel is then a combination of the values in the pixels immediately surrounding it. This means that the spatial resolution, after applying the filter, cannot in the limit be quoted as better than the width of the 3 pixel window.

4.4.1 Discrete Linear Operator

Linear systems theory provides the mathematical basis for certain filters used in digital image processing. The following discussion is derived from references [15] and [1]. A system S is considered a black box with an input $f(x)$ and output $g(x) = S(f(x))$;

$$f(x) \rightarrow S \rightarrow g(x) \quad (4.16)$$

$f(x)$ is the original image, represented in one dimension for simplicity, $g(x)$ the filtered output image, and S the filtering operation. If the filter satisfies certain conditions (if it is linear and shift invariant), then the output of the filter can be expressed mathematically in the simple form;

$$g(x) = \int f(t)h(x-t)dt \quad (4.17)$$

where $h(t)$, called the point spread function or impulse response, is a function that completely characterises the filter. The integral expression is a common form called a convolution integral.

In the digital case the integral becomes a summation;

$$g(i) = \sum_{-\infty}^{+\infty} f(k)h(i-k) \quad (4.18)$$

Although the limits on the summation are infinite, the function h is usually zero outside a given region [15].

The simplest example of a smoothing filter is one that employs spatial

neighbourhood averaging and may be formulated as a linear operation on the input image. The smoothing operator for the case of an $n \times n$ "window" may be formulated as [1]

$$g(x, y) = \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} h_{sm}(i, j) f(x + i, y + j) \quad (4.19)$$

This indicates that the output $g(x, y)$ at the point (x, y) is given by a weighted sum of input pixels surrounding (x, y) where the weights are given by $h_{sm}(i, j)$. To create the output at each successive pixel the function $h_{sm}(i, j)$ is shifted by one, and the weighted sum is recomputed. The full output is created by a series of shift-multiply-add operations, and this is called a digital convolution. [15]

In the case of $n = 3$, the output image is the convolution of the 3×3 filter kernel;

$$\begin{array}{ccc} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{array} \quad (4.20)$$

[1]

4.4.2 Nonlinear Operator

The theory of linear filters is well understood [11]. However, the properties of these filters can be inferior to a wider class which, although defying detailed theoretical analysis, can give superior results in some applications. For example, the averaging operator reduces image noise but also blurs edges. Another undesirable feature of the averaging filter is that it spreads the effect of spike noise over adjacent pixels rather than removing it. These are undesirable side effects and spawn a search for alternative smoothing approaches. Niblack [15] describes a number of filters aimed at smoothing without such effects. These include the mode where a pixel is replaced by its most common neighbour, the k nearest neighbour where a pixel is set to the average of those within a certain intensity difference k , the Sigma filter which derives k from the standard deviation of the intensity distribution, inverse gradient

filters in which neighbouring pixels are included in an average with a weighting inversely proportional to their difference from the central pixel, and the Superspike algorithm which performs a local averaging at each pixel, but uses the global image histogram to select the pixels to include in the averaging. Filters may also be used which incorporate application specific knowledge about the images being dealt with. The Median filter has proved a highly successful filter, and deserves further study in application to fringe analysis.

4.4.2.1 Median Filters

Median filtering is a nonlinear filtering technique that has been successfully applied to many signal and image processing tasks, including interferometric fringe analysis. Most notably, the median filter has been observed to be very effective for removing noise, especially impulse noise from one and two dimensional signals while satisfying the usually conflicting goal of preserving information-bearing edges. It is clear that if noise reduction is to be effected prior to detecting edges, then the filtering strategy used must not severely degrade the edge content in the image [17].

In fringe analysis edge detection usually follows pre-filtering of wrapped phase maps. It is important to retain edge location. It is also important to reduce the distortions of the filtering process, as the measurement parameter is encoded in the intensity of the interferogram. The median filter does not introduce any new intensity values to the field, but does significantly change the texture of the image. The window size of the filter is important. The data format of the original image is sufficient to store the filtered version. This is not the case with the averaging filter, for example, which introduces values which can be intermediate between those in the original field.

In order to record the averaged values of intensity, a greater intensity resolution may be required than for the original image. This is a particularly important consideration if multiple passes of such a filter are applied. Multiple passes of filters with small kernel sizes are widely employed in the realm of interferometric fringe analysis. The fringe analysis software, developed in the course of this work, expands the image intensity resolution to 16 bits, in order to lessen filtering problems of this nature. A number of iterations

of either a 3×3 average or median filter may be selected as pre-filtering options within the package.

Median filters are a subset of the class of *rank* filters where the output image intensity at a spatial location is chosen on the basis of the relative rank or intensity of pixels in the neighbourhood of the point. These filters have been widely used in fringe analysis, again for removing noise, but also for such applications as finding fringe centres [18]. Given a set of N pixel intensities obtained over a local image region, denoted simply as $f_i, i = 1, 2, \dots, N$, an ordering of these values in increasing value, i.e.,

$$\{f_1, f_2, \dots, f_N\} \quad (4.21)$$

where

$$f_i \leq f_{i+1} \quad (4.22)$$

is computed. If m indexes the median intensity, then f_m is the pixel intensity in the ordered sample set that is greater than $(N - 1)/2$ of the samples.

4.4.2.2 Median Filter Properties

The median filter has a number of interesting properties.

- i) The median filter reduces the variance of the intensities in the image. Thus, the median filter has a capability to significantly alter image texture.
- ii) Intensity oscillations with a period less than the window width are smoothed. This property is significant when considering multipass implementations of a fixed size median filter. In general, regions unchanged by the filter in a given pass are left unchanged in future passes.
- iii) Given a symmetrical window shape, the median filter preserves the location of edges.

- iv) In the application of a median filter, no new grey levels are generated. Binary images remain binary, and the dynamic range of a median filtered image cannot exceed that of the input image.
- v) The shape for a median filter may affect the processing results [1]
- vi) The median filter is nearly optimal for suppressing noise which is characterised by a large percentage of outliers, including impulsive noise [17].
- vii) Median filters with cross and X shaped geometries generally afford better results when applied to images containing a majority of horizontal/vertical and diagonal edges, respectively, while square-shaped median filters yield edge maps which are generally smoother, but with larger amounts of edge displacement when all orientations are considered [17].
- viii) Median prefiltering can improve the performance of zero-crossing type edge operators, as well as more conventional gradient based edge operators [17].

4.5 Example of Smoothing on a Speckle Image

4.5.1 Speckle

Without wishing to go into specific applications at this stage, it would be advantageous to give an example of smoothing on a typical image of the type dealt with in this work.

Speckle is a stationary interference pattern created by spatially coherent light scattered from a lambertian surface. This pattern is granular in nature. At the surface the granules are exceedingly small, and increase in size with distance. At any location in space the resultant field is the superposition of many contributing scattered wavelets, scattered from points on the surface due to its roughness. These must have a unique relative phase relationship

determined by the optical path length from each scatterer to the point in question, if interference is to be sustained [19].

4.5.2 Electronic Speckle Patterns

The field of Electronic Speckle Pattern Interferometry (ESPI) deals with capturing such speckle interference patterns in real time using devices such as CCD cameras. The resultant images consist of an array of sparse points of varying intensity, broadly describing the fringe field.



Figure 4.12: Section of an Electronic Speckle Pattern (Normalised Intensity)

Figure 4.12 shows a section of a speckle field before smoothing. The



Figure 4.13: Section of an Electronic Speckle Pattern (After 3 X 3 Average)



Figure 4.14: Section of an Electronic Speckle Pattern (After 3 X 3 Median)

intensities have been normalised, that is scaled to cover the available range, in order to make the image visible. The original image was kindly supplied by Rover's research facility at Gaydon. Figure 4.13 and 4.14 show the image after passage through a 3×3 averaging filter and a 3×3 median filter respectively. In fringe analysis a number of passes of the selected filter are usually applied to maximise the signal against noise. Towers [20] describes a method of calculating the optimum number of passes for a given spatial filter in a phase stepping system.

Some fringe analysis techniques require a Fourier transform of the image as part of the decoding process. In such a situation any filtering operations would more logically be applied in the frequency domain, as this would then represent a relatively small overhead on top of the initial transform. Convolution in the spatial domain is equivalent to simple multiplication in the Fourier domain. Other analysis techniques such as the Quasi-heterodyne method do not require an initial Fourier transform of the image, hence spatial filtering is a better option. The quantitative effects of spatial filters like the median are difficult to define as they lie outside of the simple discrete linear operator class.

4.6 Edge Detection

Edge detection is required in a whole range of image processing applications and lower level image processing algorithms such as segmentation. In fringe analysis the edges of fringes in wrapped phase maps need to be found in order to unwrap phase.

In the course of this work it has been determined that the implementation of an able edge detection technique, is an important element in any fringe analysis system. This subject is one that has been seriously neglected by many practitioners of fringe analysis. The use of primitive ad hoc methods is widespread. One of the major reasons behind this is a lack of expertise in the area of image processing, as interferometry has not traditionally been related to this subject. The need to develop an expertise in this area is only now being realised.

For example one of the more advanced phase unwrapping strategies, the

cellular automata approach, incorporates the edge detection strategy within the automaton logic. This might at first seem elegant, but the strategy is based on a search for a simple step change, which is bound to miss important edges from time to time as they sink beneath the detection threshold. Needless errors are introduced by a sub-standard edge detection technique. Even with the additional tests for 2×2 inconsistencies described by Spik [21], these errors are not necessarily detected.

Edge enhancement filters are the opposite of smoothing filters. Smoothing filters are low pass filters, edge enhancement filters are high pass. Their effect is to increase the prominence of edges in an image. However, for many applications algorithms need to produce a verdict as to whether an edge is present in a given pixel rather than merely producing a visually enhanced image. From the high pass filtered image, this may be achieved in part by a thresholding operation. In such an image, a high intensity at a pixel indicates a good confidence that an edge exists. Thresholding above a certain intensity gives an image which, if the threshold is chosen correctly, contains edge pixels only. However, the result is not guaranteed. There are some sophisticated enhancements to this basic approach.

Two filters that are often applied, as convolutions, are the Gradient and Laplacian. They are related to the vector gradient and scalar Laplacian in calculus. The gradient is defined as,

$$\nabla f = \frac{\partial f}{\partial x} \vec{i} + \frac{\partial f}{\partial y} \vec{j} \quad (4.23)$$

where \vec{i} and \vec{j} are unit vectors in the x and y directions. The Laplacian is defined as,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad (4.24)$$

A local edge is a small area in an image where the local intensity levels are changing rapidly in a simple way. An edge operator is a mathematical operator (or its computational equivalent) with a small spatial extent designed to detect the presence of a local edge in the image.

The gradient operator when applied to a continuous function produces a vector at each point whose direction gives the direction of maximum change

of the function at that point, and whose magnitude gives the magnitude of this maximum change. A digital gradient may be computed by convolving two windows with an image, one window giving the x component of the gradient, and the other giving the y component. [15]

Common masks are the Sobel and Prewitt operators, shown below;

$$\begin{array}{ccc} & -1 & 0 & 1 \\ sobel_x = & -2 & 0 & 2 \\ & -1 & 0 & 1 \end{array} \quad \begin{array}{ccc} & -1 & -2 & -1 \\ sobel_y = & 0 & 0 & 0 \\ & 1 & 2 & 1 \end{array} \quad (4.25)$$

$$\begin{array}{ccc} & -1 & 0 & 1 \\ prewitt_x = & -1 & 0 & 1 \\ & -1 & 0 & 1 \end{array} \quad \begin{array}{ccc} & -1 & -1 & -1 \\ prewitt_y = & 0 & 0 & 0 \\ & 1 & 1 & 1 \end{array} \quad (4.26)$$

Another set of masks, called the Roberts' operators, are not oriented along the x and y directions, but are similar none the less. They are defined over a 2×2 window as;

$$\begin{array}{ccc} mask_1 = & 1 & 0 \\ & 0 & -1 \end{array} \quad \begin{array}{ccc} mask_2 = & 0 & 1 \\ & -1 & 0 \end{array} \quad (4.27)$$

The Laplacian operator, is also computed by convolving a mask with the image. Second derivative operators, such as the Laplacian, use zero crossings of the second derivative to detect step edges, see Figure 4.15. However, there are a number of problems associated with this approach. Higher order derivatives exacerbate noise, and zero crossings often occur in the absence of step edges. An example is the case of a corrugated surface. The Laplacian is seldom used by itself for edge detection as it is unacceptably sensitive to noise, the Laplacian-of-a-Gaussian approach, in which it is combined with a smoothing function, is much more effective. This technique is described later.

Since edges are a high-spatial-frequency phenomenon, edge detectors are usually sensitive to high-frequency noise.

There are a large number of edge operators. Their number makes them

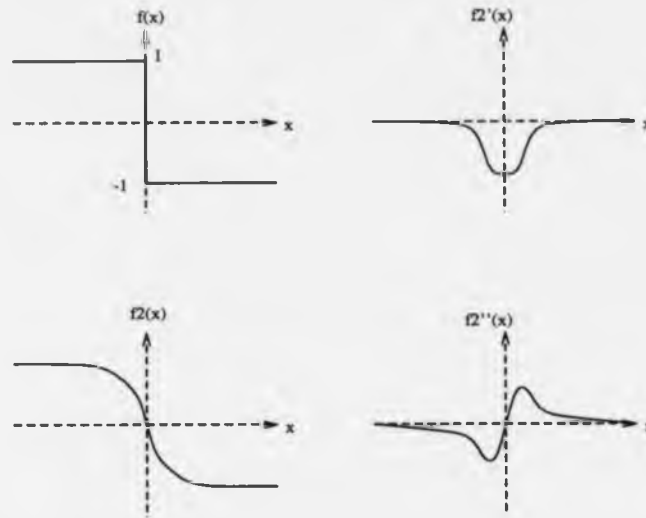


Figure 4.15: Illustration of Step Edge Detection using Zero Crossing of 2nd Derivative, ($f(x)$ is a step function, $f2(x)$ is a smoothed step function)

very difficult to compare and evaluate. For example, some operators may find most edges but also respond to noise; others may be noise-insensitive but miss some crucial edges.

Various strategies have been developed for measuring the efficiency of edge detectors. For example, reference [16] describes a method due to Pratt, and gives a graph showing the performance of the Prewitt/Sobel operators against the Roberts. The figure of merit is defined as

$$F = \frac{1}{\max(N_A, N_I)} \sum_{i=1}^{N_A} \frac{1}{1 + (ad_i^2)} \quad (4.28)$$

where N_A and N_I represent the number of actual and ideal edge points, respectively, a is a scaling constant, and d is the signed separation distance of an actual edge point normal to a line of ideal edge points. The term ad_i^2 penalises detected edges which are offset from their true position; the penalty can be adjusted via a .

Figure 4.16, shows typical curves for different edge operators. It was made using a step edge and the threshold for reporting an edge was chosen independently for each operator so as to maximise Eqn. 4.28. Pratt defines

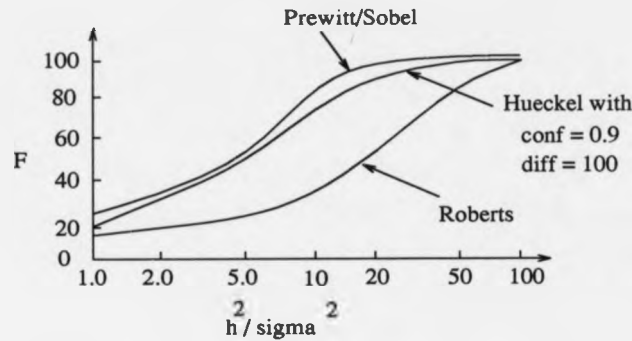


Figure 4.16: Typical Curves for Different Edge Operators

a signal-to-noise ratio as the square of the step edge amplitude divided by the standard deviation of Gaussian white noise $\frac{h^2}{\sigma^2}$.

Such comparisons can provide a gross measure of differences in the performance of operators even though each operator embodies a specific edge model and may be best in special circumstances. Such measures are not perfect however. Reference [5], gives an illustration of an instance where Pratt's measure fails to give a reasonable result. That is where the figure of merit is high, but upon visual inspection edges appear broken. The reason underlying this is that Pratt's figure does not take into consideration any information on the distribution of the edge points along the edge. An example is given of the same edge being detected twice, on either side of the true position, but because in both cases the detected points are close to the true position of the edge, the estimates both increase the figure of merit.

Enhancement/thresholding based edge detectors such as the Roberts' are still widely used, but in many applications more sophisticated edge detectors, modeled with characteristics found in some mammalian visual systems, are being used. For example the Marr and Hildreth [6] Laplacian-of-a-Gaussian ($\nabla^2 G$) or (LOG), which is similar in operation to the post-retinal ganglion cells of the eye.

The LOG approach, in essence, attempts to reduce the effect of noise by smoothing before edge enhancement. The LOG operator smooths the image through convolution with a Gaussian-shaped kernel, see Figure 4.17;

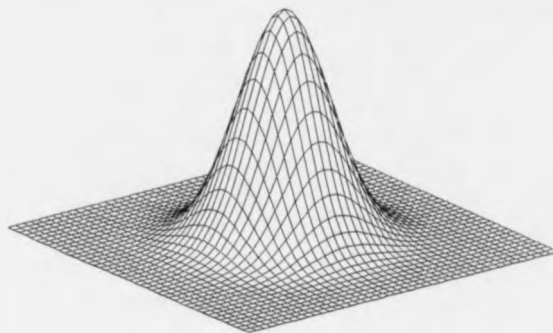


Figure 4.17: Gaussian Smoothing Kernel (Sigma = 5)

$$G(x, y) = \left(\frac{1}{2\pi\sigma^2} \right) \exp\left(\frac{-(x^2 + y^2)}{2\sigma^2} \right) \quad (4.29)$$

$G(x, y)$ has circular symmetry. The smoothing effect may be varied through the parameter σ . Canny [22] gives a good account of why a smooth projection function such as the Gaussian should be used.

When a linear operator is applied to a two-dimensional image, a weighted sum of the input values is formed. In edge detection this sum will be a difference between local averages on different sides of the edge. This output represents a kind of moving average of the image. Ideally an infinite projection function would be best, but real edges have a limited spatial extent. It is therefore necessary to window the projection function. If the window function is abruptly truncated, for example if it is rectangular, the filtered image will not be smooth because of the very high bandwidth of this window [22]. That is lower frequencies which are not edges could be picked up as edges.

The Gaussian kernel guarantees zero crossings of the second derivative are preserved. Following Gaussian smoothing, the Laplacian operator is applied. The resulting operator has high-frequency emphasis characteristics. The kernel is usually applied at two or more scalings. The scaling of the kernel is

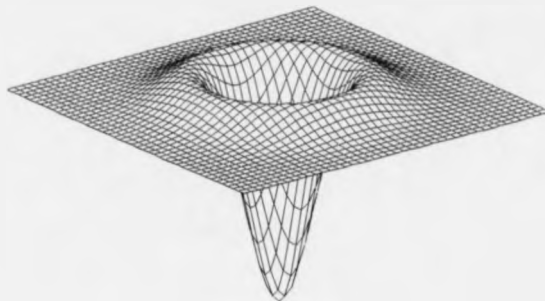


Figure 4.18: Laplacian-of-a-Gaussian Smoothing Kernel (Sigma = 5)

related to the spatial extent of the edge to be detected. By using more than one scaling of kernel, in neighbouring frequency bands, edges that appear at the same spatially localised position gain confidence as representing true physical edges. The operator is given below and shown in Figure 4.18;

$$\nabla^2 G(x, y) = \left(\frac{1}{\pi \sigma^4} \right) \left(\frac{x^2 + y^2}{2\sigma^2} - 1 \right) \exp\left(\frac{-(x^2 + y^2)}{2\sigma^2} \right) \quad (4.30)$$

Because the resultant zero crossings are of primary significance, the constant term $\left(\frac{1}{\pi \sigma^4} \right)$ is often replaced with an arbitrary scaling factor. The operator is often plotted with a negative scaling, in this case it resembles a Mexican hat and is therefore referred to as a Mexican hat operator. The LOG operator, with two kernel sizes, has in the past been approximated by the difference of two images convolved with the Gaussian operator alone.

It is comparatively difficult, using the LOG operator, to determine the magnitude of an edge, that is how confident the detection has been. This is due to the technique using the second derivative which indicates the presence of maxima or minima, but not their magnitude. The Canny edge detection technique [22] uses the 1st derivative and is so better suited to estimating the strength of edges.

An adaptive thresholding scheme is described with relation to the Canny

technique. Two thresholds for the magnitude of the edge are set, a low value and high value. If the low value were used alone it would detect most edges, but also detect noise points. To avoid this the high threshold is used.

Edge detection is performed on the image, recording points whose magnitudes fall above the low or high thresholds. The edge image produced is then reprocessed as described below.

- i) For all the pixels which produced an edge magnitude above the low threshold, and are not confirmed edge points, do ii).
- ii) Compare this point with its adjacent positions in the edge image. If any adjacent pixel (including diagonals) is an edge point with a magnitude above the high threshold (or a confirmed edge point) then confirm this point as an edge point. If this point is confirmed goto ii) otherwise continue with step i).

Canny claims that a one-dimensional LOG edge detector is almost identical to his own but that in two dimensions the directional properties of his operator enhance its detection and localisation performance against the LOG. The adaptive thresholding idea is an interesting one, but the Canny edge detection technique is computationally intensive, compared to simple gradient operators.

In the Interferometric application, which is covered extensively elsewhere, the adaptive thresholding technique is applied to a simple gradient operator (the Sobel). This has yielded a fast and effective edge detector, well suited to the problem found in fringe fields. That is, the 2π phase jump in wrapped phase maps is not always adequately defined. It can fluctuate across the field. In some cases it would pass undetected by a fixed threshold edge detector. The adaptive technique reduces the problem.

Such an intensity fluctuation represents a distortion of the parameter being measured, but this can be corrected if it is prevented from impairing the operation of the image processing functions. These aim to unwrap the phase of the fringe field and present it as a near continuous 2D function. It is much easier to post-process a near continuous field, in order to correct the fluctuation, than a wrapped one.

Because the phase change is very sharp, it is limited to a small spatial area, which means that a relatively small window operator may be employed.

A large and sophisticated window operator, employing gaussian smoothing etc, although effective, does not on balance seem to be the best choice. This is especially the case when considering that spatial smoothing is normally applied to the input interferograms before the edge detection process to reduce noise. That is, it would often be unnecessary to apply yet another smoothing operation during the edge detection phase.

An example wrapped phase map, upon which edge detection has been performed, is shown in Figure 4.19. The result of a Sobel edge detection with adaptive (hysteresis) thresholding is shown in Figure 4.20. The grey lines indicate where edges have been found above a low threshold *and* which were connected to edges found above the high threshold, which are shown in black. If the Sobel were used in its standard mode, the edges denoted by the black lines alone would be found.

The result of a single pass of the Laplacian-of-a-Gaussian zero crossing edge detector is shown in Figure 4.21, for comparison. The kernel size for the LOG operator was specified as 9 and σ as 1. The zero point in the figure is mid way along the grey scale range, so the zero crossings, and therefore the edge points, are represented by a black/white or white/black transition (similar to the original wrapped phase map in fact).

4.7 Industrial Requirements for a Fringe Analysis System

There are some industrial applications which require real time analysis, for example TV holography (ESPI) on the production line. However, there are many others for which this is not the case, for example holographic flow visualisation. In order to gain speed, dedicated hardware or multi-processor systems might be employed. However, the cost of a dedicated piece of hardware or multi-processor computer, may not be thought economic for the application, both in terms of the speed with which results are required, and also in respect of the difficulty in updating such systems.

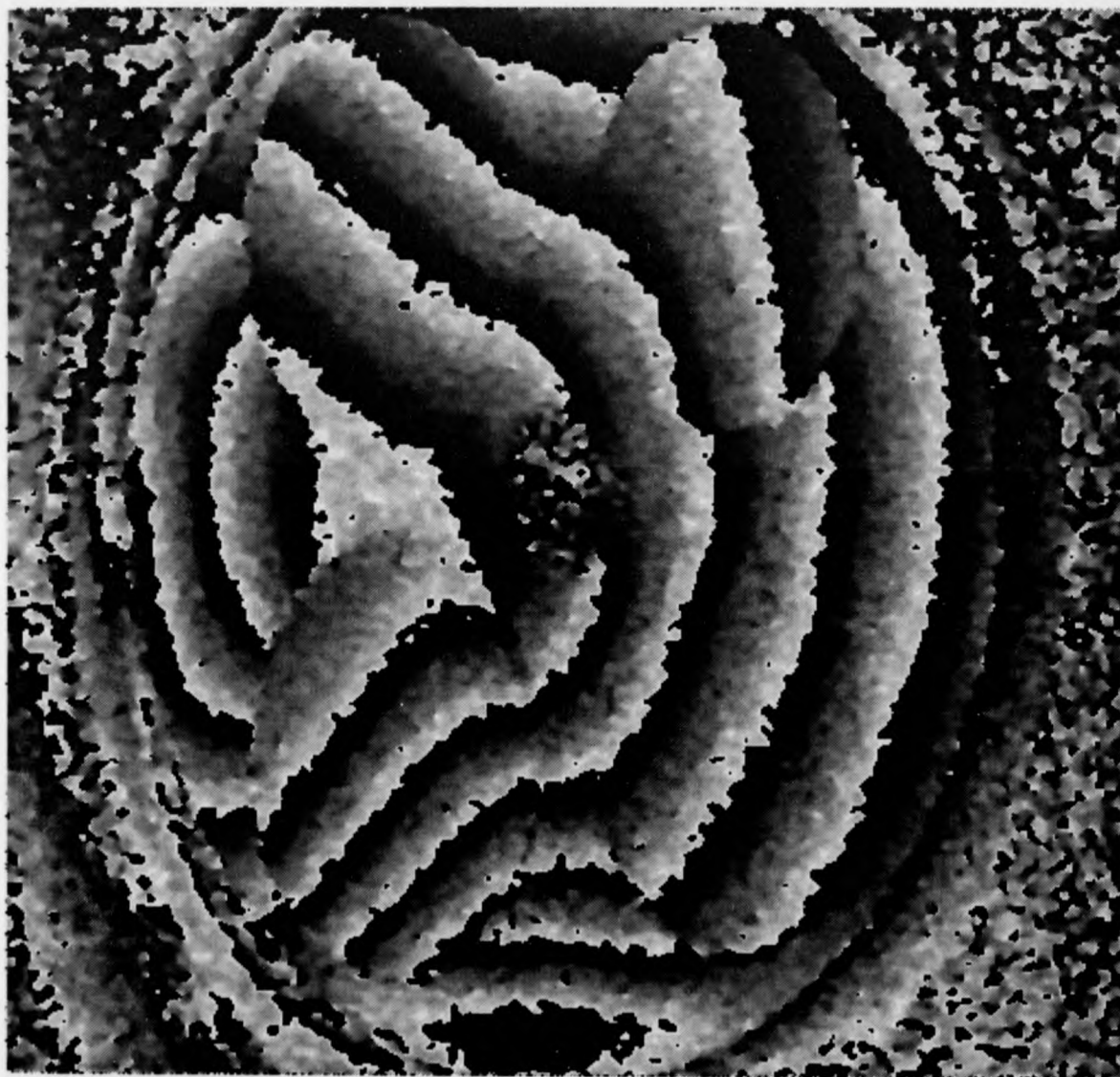


Figure 4.19: Example Wrapped Phase Map for Edge Detection

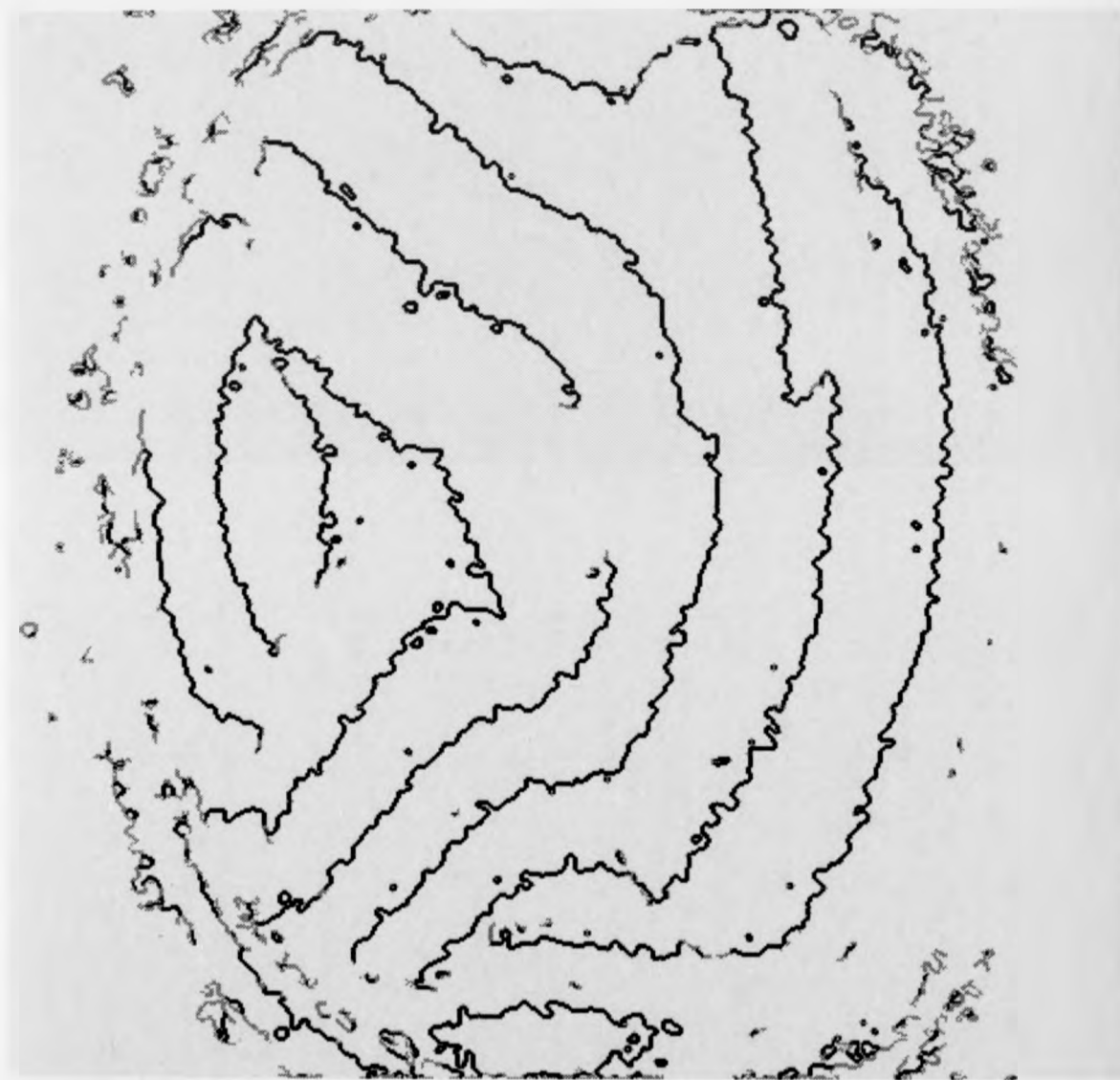


Figure 4.20: Sobel Gradient Edge Detector with Adaptive Thresholding Applied to Wrapped Phase Map, Kernel = 3 X 3



Figure 4.21: Laplacian of a Gaussian Edge Detection Applied to Wrapped
Phase Map, Kernel = 9 X 9, Sigma = 1.0
186

As an alternative to more dedicated systems there has been a growth in the market for general co-processors. For example, the powerful i860 processor can be supplied on a PC-card and yet remains a flexible unit, capable of compiling standard C or Fortran code. For some industrial applications batch processing is a possibility. That is, image data may be captured in real time and recorded for processing later. This batch processing may take an arbitrary period. To take another scenario, it might be envisaged that a measurement is only required intermittently, thus allowing processing to be carried out in the intervening intervals.

Yet another option is to employ a powerful remote networked mainframe or workstation, to perform the analysis task. This permits, for example, the main computing facility of an industrial site to take on the computational burden. This approach has been applied here. The network of departmental Sun Sparc processors has been accessed over an ethernet connection from a data capture PC. The PC Net Filing System software package (which is a piece of system software for PC-Compatible computers) and an ethernet card (installed in the PC), permits a PC computer to communicate with other machines on the network at high speed[7].

4.8 System Development by Other Researchers in Fringe Analysis

Researchers have explored the efficient implementation of the phase stepping algorithm in hardware [10], despite the limits upon flexibility that such systems impose. A number of workers in fringe analysis are using dedicated systems comprising a hardware and software element [8, 9, 21]. Breuckmann has done so in an attempt to commercialise the technology. Buckberry has built an ESPI system for investigating automotive structures, requiring on line processing in the optics lab at Rovers's research facility at Gaydon, employing a frame store with associated processing capabilities. Spik has been encouraged by Ghiglia's speculations [23] to implement the Cellular Automata phase unwrapping strategy upon an array processor. The author has selected a software implementation because of the advantages of devel-

opment flexibility, portability and extensibility.

It is difficult to update hardware dependent implementations, this will be necessary as developments continue in imaging technology. Kujawinska appears to be using a flexible workstation for her work[24].

Some of the software+hardware systems above are unable to deal with images larger than the size of the dedicated frame store, or the memory of the frame processor. This is a limitation. For example a flat bed scanner, (most often used in desk top publishing applications) is able to produce high quality images at typical resolutions of 2500 by 3500 pixels with 8 bits intensity resolution (for example the Hewlett-Packard Scanjet Plus at 300 dots per inch). The applications to which the scanners can be applied are limited, as there is the limit that the frame must remain stationary for the duration of the movement of the linescan element (around 15 seconds). However, the scanner is an inexpensive device (about £1000), is widely available, and is representative of the resolutions that future devices will offer with additional flexibility. CCD cameras are available with resolutions of 4096 by 4096 pixels (by Kodak), although as yet, these are expensive (about £20,000), and have a read out time of several seconds.

The software solution presented is able to deal with arbitrary sized images, including those generated by high resolution CCD cameras and scanners. Processing speed, under this regime, is to be gained from the on going development of high speed sequential processors.

Breuckmann gives execution times for his hardware, which employs an 'FFP-16' processor. As the system is a commercial one the analysis method is not revealed. The images which have been shown at conferences do not contain any large scale discontinuities. The capabilities of the analysis system are therefore unknown.

Using this hardware the complete analysis is performed in about two and a half seconds, Figure 4.8.

Figure 4.8 shows the performance of the FRAN system on a variety of processors for a similar Phase Stepping analysis. However, prefiltering is not applied and only three images are used in the analysis. The complexity of the phase unwrapping algorithm differs also. Nevertheless the frame processor is only 30 times faster than the general software solution running on a SPARC

Recording of 4 fringe patterns + phase-step algorithm	200 msec
Fringe enhancement by optimised filter transformations	200 msec
Detection of local inhomogeneities and faults	200 msec
Problem oriented demodulation of fringe patterns	1-2 sec
Calculation of phase maps in colour-coded fringe patterns	video real time

Table 4.5: Execution Times for Breuckmann's Dedicated Software+Hardware Combination

Machine	Phase Unwrap by Fringe Counting	Phase Unwrap by Minimum Spanning Tree
PC-386 (25Mhz+FP)	7 mins 30 secs	11 mins 21 secs
Sun 3/80 workstation+FP	6 mins 43 secs	10 mins 5 secs
Sun SPARC station 1	1 min 22 secs	2 mins 2 secs
Sun SPARC station 2	38 secs	59 secs

Table 4.6: Execution Times for Hardware Independent Analysis Software, on Various Processors, Using 3 Phase Stepped (512 by 512) Images, Without Prefiltering

station 2.

4.9 The Development Process for the FRAN Fringe Analysis System

This section describes the approach taken to the development of the fringe analysis system. The goal of this system was to enable the accurate, automatic and rapid transformation of interference patterns into a continuous format, that non-experts could interpret and combine with other measures. The system required the development of the automatic phase unwrapping algorithm, as a research activity.

It is inadvisable to confuse algorithmic development with architecture or hardware specific issues too early on. The task of algorithm development is hindered if optimisation of a particular software/hardware combination is attempted in the course of development.

The test cycle inherently requires modification of the system in the search for improved performance. The prototype system has been implemented in

software. The aim has been to maintain generality, modularity, and clarity in programming the prototype where possible to ease the modification process.

The development process has been eased by a flexible development environment, with few hardware imposed limits. This environment was provided by the Sun Network of the Engineering Department, at Warwick.

The C programming language has been employed. The system is therefore portable to other processors which support this language. The software has been tested on a variety of machines, PC-386, T-800 Transputer, Sun 3, Sun 4 (Sparc) and an i860 card.

Having assessed the requirements of the Minimum Spanning Tree phase unwrapping algorithm, it is seen that a parallel implementation would best match its structure. It will be shown later that an SIMD processor array, consisting of specialised processors optimised for MST computation could provide a high performance system.

A brief review of high speed sequential RISC processors is given below, as these processors currently provide the most cost effective platform for a fringe analysis system.

4.10 High Speed Sequential Microprocessors

Reduced Instruction Set Computers/Microprocessors (RISC) have become prominent in recent years, for both use in sequential and parallel processing systems. These processors have a small number of highly optimised instructions.

As specific examples, the Sparc processor from Sun Microsystems (used in their Workstations), the i860 from Intel, and the Transputer from Inmos are all RISC processors.

A brief outline of the capabilities of the i860 processor, and so current technology, is given below.

The Intel i860 64-bit microprocessor is a general purpose microprocessor integrating an integer RISC core unit, a floating point unit, a paged memory management unit, instruction and data caches, and 3-D graphics software assist logic in a single VLSI component. The versatile 64-bit design of the i860 microprocessor balances performance across integer, floating point,

and graphics processing capability. Its parallel architecture achieves high throughput with RISC design techniques, pipelined units, wide data paths, large on-chip caches and fast one micron CHMOS IV silicon technology [25].

The i860 performs at up to 40 Mhz, with a peak performance of 80 million floating-point operations per second (MFLOPS) [25], its sustained performance is however considerably less than this, as the figure requires that the pipe line is kept constantly filled, the system in which the processor is employed also has a marked effect on performance. The system to which the author has had access was resident as a card in a 386 PC, the PC bus acted as a bottle neck to data transfer from the PC disc. The Sun Sparc station 2 is quoted as having a peak performance of 4.2 MFLOPS, with a clock speed of 40 Mhz.

However, the processor itself is not sufficient to guarantee performance. It requires the backup of a reliable operating system, high speed peripherals (disc etc), and graphical support. In the case of the Sparc these are all found within the Sparc station 2.

4.11 Parallel Processing

Computers are conventionally thought of as doing one thing at a time. In fact parallelism has existed in computer systems in one form or another for decades and has always been on the increase. This can be seen in the upward growth of the word length of Microprocessors, (each bit is processed in parallel) and in the form of intelligent peripheral devices, such as disc controllers, which perform their tasks whilst the processor gets on with something else.

Doing things in parallel, providing communication overheads are small, is faster than doing things sequentially. There has been keen interest in parallel multiprocessor machines over the last decade. However, there has also been a great improvement in the power of individual microprocessors such as those mentioned above, which exploit parallelism at a lower level, through pipelining and on chip coprocessors for example. This increase in the power of individual processors, has reduced the market for multiprocessor machines, as clearly the single processor may continue to be employed in traditional architectures and execute traditional sequential codes.

There comes a point, however, when the problem size exceeds the capability of the sequential processor. In image processing, this point is fast approaching, if it has not already been reached. In fringe analysis this is particularly the case, considering the current state of video technology which promises much increased resolution.

Although a hardware independent sequential source code has been evolved, it is also relevant to point out the advantages of alternative strategies, involving parallelism.

The first step in a parallel approach to the solution of any problem is identification of parallelism within the problem. From this point, an algorithm utilising the parallel elements may be established (or one could say that the problem may be mapped to a parallel algorithm, exploiting parallelism inherent in the problem).

Implementation upon a specific parallel architecture is then considered. In order to optimize this mapping, parallelism in the algorithm must be mapped to the parallel capacities of the machine. That is there are two mapping procedures, see Figure 4.22.

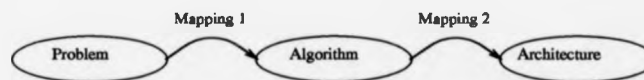


Figure 4.22: Mapping of Problem to Algorithm, Algorithm to Architecture

Later on it will be seen how the parallel minimum spanning tree algorithm has been developed for the Parallel Random Access Machine (PRAM) model of parallel computation, which is a general purpose target architecture for use in developing parallel algorithms, and how subsequently this algorithm may be implemented efficiently on a specialised SIMD (see below) processor array. The development of this processor suggests one course for the realisation of a parallel fringe analysis system. The various elements of such a system are briefly considered with respect to parallelisation.

There are two types of parallel processing elements. These are Single Instruction, Multiple Data (SIMD) processors and Multiple Instruction, Multiple Data (MIMD) processors. The sequential computer is referred

to as a Single Instruction Stream, Single Data Stream machine or (SISD) processor.

SIMD refers to a computer architecture in which each node has local memory but operates in lockstep with the same global instruction scheme as every other node. This is a more general approach than one might think at first, since the common instruction may lead to different results through the dependence on the data stored at each node. SIMD works best when vector instructions work best. For example in dealing with arrays in for-loops. Hence, to have the opportunity for massive parallelism in SIMD there must be massive amounts of data, or data parallelism. SIMD is at its weakest in 'case' or 'if' statements where each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue.

MIMD refers to a machine architecture in which each node operates independently on its own local instruction stream and data. The node of an MIMD machine may itself consist of an MIMD or SIMD collection of smaller subunits [26].

4.12 Fringe Analysis in Parallel

The fringe analysis problem may be broken down into several phases.

- i) Prefiltering of the interferogram(s).
- ii) Computation of the wrapped phase map.
- iii) Edge detection.
- iv) Phase unwrapping.

Each of these phases has a parallel element which may be exploited in the development of a parallel system. The image processing functions are largely localised spatially, that is independent of global image data. The central approach is then to spatially decompose the interferogram into small image sections, and allocate each area to a distinct processor or set of processors. This approach is particularly suited to the tiling approach. Each tile may

be processed in parallel with adjacent tiles, from the prefiltering stage to delivery of the unwrapped phase map. Parallelisation of the various analysis phases is considered below.

i) Prefiltering of the interferogram(s).

Filtering may be performed either spatially or in the frequency domain. Spatial prefiltering may be performed by simple spatial decomposition of the field. A relatively small amount of communication would be necessary between adjacent nodes, to transfer data concerning pixels along adjacent boundaries.

Filtering in the frequency domain is less simple to parallelise, as the transform to the frequency domain requires access to global image data.

ii) Computation of the wrapped phase map.

The phase stepping method of obtaining a wrapped phase map is ideal for parallelisation by spatial decomposition. Each wrapped phase value is independent of the data in surrounding pixels. No node communication would be necessary.

The Fourier Transform method of fringe pattern analysis is less simple to parallelise. As mentioned above the transform to the frequency domain requires access to global image data.

iii) Edge detection.

Simple edge detection algorithms, based on edge operators, may be parallelised fairly simply. Some communication with adjacent nodes may be necessary to communicate pixel data at the border of the edge operator. However, as the edge operators employed in fringe analysis have a small kernel size, this would not present too much of an overhead.

iv) Phase unwrapping.

Spik has implemented the cellular automata method of phase unwrapping in parallel [23]. The implementation has employed a

Linear Array Processor (LAP). The device has been employed as a peripheral of a PDP11 sequential computer. This seems to be a SIMD (Single Instruction Multiple Data) device with 256 separate processors. Each processor performs the same instructions on different data points. The control software permits the device to be treated as a 2-D array of processors, which can make simple calculations for each cell of the memory using neighbouring cell values. The image is transferred from a frame capture board to the LAP memory raster-by-raster, processed and transferred back. The process may occur a significant number of times during execution of the algorithm, due to a limit of the processing system.

Spik describes the limitations of the LAP processor in terms of the word length. The limitations of the 8-bit LAP memory impose specific conditions on data preparation. Data for the LAP must be positive in the 8-bit range, that is from 0 to 255. This is sufficient resolution for one phase-fringe, but during the unwrapping process the required range expands. A scaling process is therefore employed which causes a decrease in phase resolution. [23]

Spik speculates that the low resolution solution obtained, may be compared with the original wrapped phase map and used to unwrap it using the original phase values. This is all very well but the ability of the algorithm to detect errors in the phase map is much reduced because of the small dynamic range, during execution of the phase unwrapping algorithm. Edge detection capability is impaired.

Spik states that the cellular automata routine executing upon the NPL LAP is able to perform five local iterations in one pass through the data, but the routine is still slower than a path dependent routine sequentially processing phase data. This was as a result of the need to transfer data a number of times from the frame store to the processor array, a limitation of the specific

processing system. The following relationships are given, for a path-dependent algorithm

$$T_0 = t.m^2 \quad (4.31)$$

where m is the array resolution and t is the time to process a single pixel. For a cellular automata algorithm the total phase unwrapping time is

$$T_c < \frac{t.m.n_f}{2} \quad (4.32)$$

where n_f is the number of phase fringes in the fringe field.

The efficiency of phase unwrapping for the MSTT approach hinges on the efficiency with which the minimum spanning tree algorithm is implemented. The parallel algorithm and its realisation on a VLSI chip are discussed in the next section.

4.13 Parallel Minimum Spanning Tree Algorithm for Phase Unwrapping and its Implementation

In Computer Science, Graph theory has been used as a tool for exploring efficient parallel algorithms [28]. As it turns out trees are used as a basic structure in parallel computations, for example in the structure of computation, as a data structure or as a structure of processors. Consequently a considerable amount of work has been done into optimal parallel algorithms for the computation of graph algorithms, including minimum spanning trees.

An important feature of MST computation is that it is a closely related problem to the computation of graph connected components. This means that an efficient parallel architecture for the computation of the latter should offer an efficient solution for the MST computations as well. An algorithm

for the MST has been demonstrated by Chin & Chen [27, 28] for the Parallel Random Access Machine (PRAM) computational model. The PRAM assumes that all processors in a network have access to a common memory and that no memory time penalties are incurred. The model provides a good theoretical basis for the development of algorithms but in reality is unrealisable, without memory time penalties. The model neglects any hardware constraints which a highly specified architecture would impose. In any realisation of a PRAM there would be all possible links between processors and memory locations. This complexity of linkages is not physically realisable in present-day hardware.

Using the PRAM model simplifies discussion of efficient parallel algorithms whilst at the same time it is known that the algorithm may be implemented upon a real parallel machine and still cost polylogarithmic time [28].

The parallel MST algorithm by Chin & Chen takes $O(\log^2 n)$ time on $\frac{n^2}{\log^2 n}$ processors. The tree is formed in a hierarchical manner in an iterative process as follows.

Initially label all nodes uniquely

- i) A minimum edge is found from each node/pseudo-node to another in parallel.
- ii) The groups of nodes that are connected by these minimum edges are merged to form new psuedo-nodes.
- iii) All nodes/pseudo-nodes in the same newly formed pseudo-node are then relabelled with the same label.

Steps i) to iii) are repeated until a single pseudo-node remains which represents the MST. This requires a total of $\log n$ iterations. In the second and subsequent iterations pseudo-nodes exist which are connected by minimum weight edges, from nodes within them. A minimum weight connection between pseudo-nodes may exist between any of the pseudo-nodes constituent nodes.

It is important to develop parallel algorithms for generalised parallel computing platforms, by for example, using models such as the PRAM. The

above algorithm is a generalised one. Moving to a specific parallel architecture would normally mean some additional time complexity cost in the implementation of the algorithm. However, in the case of the MST, a novel VLSI realisation has permitted no cost penalty in the implementation of the algorithm, when compared with the PRAM model, of $\log^2 n$.

This realisation is a result of collaboration between the University of Massachusetts and Hughes Aircraft Company [29]. As a by product of the development of a multi-level massively parallel machine, an enhancement has been made to its SIMD processor which facilitates the grouping of the Processing Elements (PEs) in an arbitrary broadcast network. This network is termed a Gated Communication Network (GCN) [30]. When the MST algorithm is considered this GCN enables the arbitrary connection of nodes within each pseudo-node so that its minimum weight may be computed in parallel without the cost of data transfer between nodes, as discussed by Shu and Nash in reference [31].

Shu and Nash [31] give a comprehensive description of the parallel Minimum Spanning Tree algorithm and its implementation on the GCN. The GCN itself is described. It is embedded in a content-addressable array of 64 bit-serial processors, each having 320 bits of on-chip memory and 32 Kbits of off-chip backing store (employing 256 Kbit Video-RAM chips). The device employs more than 100,000 transistors. The 320 bits of RAM are divided into multiple pages to provide a double-buffered swapping mechanism between backing store and on-chip memory. Each PE is a bit-serial machine with a 1-bit data path to access memory. However, the first two 128-bit pages of PE memory have an 8-bit data path to support the 8-bit control registers used by the GCN. The 8-bit data path allows the GCN to be set up with one instruction instead of eight. This also allows 8-bit transfers between the backing store and on-chip memory, and improves floating point performance as shifts may be made 8-bits at a time.

Considering the phase unwrapping process, the phase image would be mapped on to the otherwise general SIMD processor array. It is assumed (for convenience) that the processor array is the same size as the image data. A graph would first be computed in place (in parallel) whose edge weights relate the confidence of alternative unwrapping routes. Each PE would then

hold a single image node, a pixel of the input phase image, and its associated four weights to adjacent nodes. Barriers would be included to, initially, prevent unwrapping through tile boundaries during the pixel level of phase unwrapping.

Image capture systems continue to provide higher and higher resolutions, it is likely that processor arrays will be smaller than the size of the images as such arrays are not increasing in size at the same rate. The image can be split into sheets containing a number of whole tiles, and each sheet computed in turn on the processor array. Given an image size of M by M pixels and a processor array size of P by P , the computation time under this scheme would be increased by a factor of $\lceil \frac{M^2}{P^2} \rceil$.

Processing is also affected by the image quality as this determines the size of tiles. If an image area contains no fringe boundaries, that is points where the phase is not wrapped, then there is no need for phase unwrapping. That is some areas need not be processed by the low level pixel phase unwrapping procedure, which in turn obviates the need to compute the MST for that area. Considering the sheet model described above, this would result in poor utilisation of the processor array. It is envisaged that a load balancing approach allocating tiles to processors could improve processor utilisation.

In terms of computation the smaller the tiles the better. That is the complexity of the parallel MST algorithm is $\log^2 n$, so the smaller n , (where n is the number of pixels in a tile) the better the performance. However, it is important that the tiles are larger than the discontinuities to be detected. There is therefore a trade off between successful phase unwrapping and performance. In practice (on the sequential FRAN system) a fixed tile size has been used to solve a variety of images, this tile size has been made large enough to deal with the majority of discontinuities across several image types. Selecting the tile size automatically on an image by image basis would improve performance by using the smallest tile size possible that also coped with the discontinuities in the image. This is for the future.

4.14 Conclusion

This chapter has dealt with a variety of image capture and processing issues. The noise level of a CCD and digitiser combination has been investigated. The characteristics of low pass filters have been reviewed. The effective detection of phase rollover points in wrapped phase maps has been considered. The options for implementation of a fringe analysis system have been considered. Parallel implementation of the MSTT algorithm has been explored on an SIMD array.

Chapter 5 gives some results for the MSTT phase unwrapping process, from a number of fringe analysis applications.

Bibliography

- [1] R. J. Schalkoff, *Digital Image Processing and Computer Vision*, John Wiley & Sons, Inc., 1989.
- [2] R. E. Benner, R. W. McClane, J. A. Dixon, R. C. Straight, "In-vitro CCD Measurements of Light Distributions from Diffusing Optical Fiber Tips used in Photodynamic Therapy", *SPIE Vol. 1202 Laser-Tissue Interaction*, pp. 103-115, 1990.
- [3] B. R. Masters, T. Bruchman, G. Q. Xiao, G. S. Kino, "Charge-coupled Devices for Quantitative Confocal Microscopy of the Eye", *SPIE Vol. 1242 Charge-Coupled Solid State Optical Sensors*, pp. 48-58, 1990.
- [4] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in C The Art of Scientific Computing*, Cambridge University Press, ISBN 0-521-35465-X, 1988.
- [5] T. Peli, D. Malah, "A Study of Edge Detection Algorithms", *Computer Graphics and Image Processing* 20, pp. 1-21, 1982.
- [6] D. Marr, E. Hildreth, "Theory of Edge Detection", *Proc. R. Soc. Lond. B* 207, pp. 187-217, 1980.
- [7] Sun Microsystems, "Network Programming Guide", Part Number: 800-3850-10, Revision A of 27th March, 1990.
- [8] B. Breuckmann, "Techniques and Applications of Online Fringe Analysis", *Applied Optics Digest*, 17th-20th September, pp. 249-250, 1990.
- [9] C. Buckberry, J. Davies, "Digital Phase-Shifting Interferometry and its Application to Automotive Structures", *Applied Optics Digest*, 17th-20th September, pp. 275-276, 1990.

- [10] H. H. Meyer, M. Goessel, "High-speed Arithmetical Support in Phase-shifting Interferometry", Fringe '89, Proceedings of the 1. International Workshop on Automatic Processing of Fringe Patterns held in Berlin (GDR), April 25-28, 1989.
- [11] A. V. Oppenheim, R. W. Schaffer, Digital Signal Processing, Prentice/Hall International, Inc., ISBN 0-13-214107-8, 1975.
- [12] Matrox Electronic Systems Ltd., Pip Video Digitizer Board, Hardware Manual, 289-MH-00 Rev. 2, February, 1987.
- [13] TIFF Specification 5.0, An Aldus/Microsoft Technical Memorandum, Developers' Desk, Aldus Corporation, 411 First Ave., Suite 200, Seattle, WA 98104, Tel. (206) 622-5500, 1988.
- [14] G. T. Reid, "Automatic Fringe Pattern Analysis: A Review," Optics and Lasers in Engineering, vol. 7, pp.37-68, 1986/7.
- [15] W. Niblack, An Introduction to Digital Image Processing, Prentice-Hall International (UK) Ltd, 1986.
- [16] D. H. Ballard, C. M. Brown, Computer Vision, Prentice Hall, ISBN 0-13-165316-4, pp. 75-88, 1982.
- [17] A. C. Bovik, T. S. Huang, D. C. Munson, "The Effect of Median Filtering on Edge Estimation and Detection", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-9, No. 2, pp. 181-194, March 1987.
- [18] K. Andresen, P. Feng, W. Holst, "Fringe Detection Using Mean and n-Rank Filters", Fringe '89, Proceedings of the 1. International Workshop on Automatic Processing of Fringe Patterns held in Berlin (GDR), April 25-28, 1989.
- [19] E. Hecht, Optics, Addison-Wesley, 1987. ISBN 0 201 11611 1
- [20] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Automatic Interferogram Analysis Techniques Applied to Quasi-heterodyne Holography and ESPI", Optics and Lasers in Engineering, 14, pp. 239-281, 1991.

- [21] A. Spik, D. W. Robinson, "Investigation of the Cellular Automata Method for Phase Unwrapping and its Implementation on an Array Processor", Optics and Lasers in Engineering, Vol. 14, No. 1, pp.25-37, 1991.
- [22] J. Canny, "A Computational Approach to Edge Detection", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-8, No. 6, pp. 679-698, 1986.
- [23] D. C. Ghiglia, G. A. Mastin, and L. A. Romero, "Cellular-Automata Method For Phase Unwrapping," J. Opt. Soc. Am., vol. 4A, pp. 267-280, 1987.
- [24] M. Kujawinska, J. Wojciak, "High Accuracy Fourier Transform Fringe Pattern Analysis", Applied Optics Digest, 17th-20th September, pp. 257-258, 1990.
- [25] i860 64-Bit Microprocessor Hardware Reference Manual, Intel Corporation, Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1990.
- [26] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors: Volume 1 General Techniques and Regular Problems, Prentice-Hall, Inc., ISBN 0-13-823469-8, 1988.
- [27] F. Y. Chin, I Chen, "Efficient Parallel Algorithms for Some Graph Problems", Communications of the ACM, Vol. 25, Number 9, September 1982.
- [28] A. Gibbons, W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.
- [29] C. C. Weems, S. P. Levitan, A. R. Hansen, E. M. Riseman, D. B. Shu, J. G. Nash, "The Image Understanding Architecture", Int. Journal of Computer Vision, Vol.2, pp. 251-282, 1989.
- [30] D. B. Shu, J. G. Nash, "Finding Connected Components of a graph on a Gated Connection VLSI Network", J. VLSI Signal Processing, to be published.

- [31] D. B. Shu, J. G. Nash, "Minimum Spanning Tree Algorithm on an Image Understanding Network", SPIE Vol. 939, Hybrid Image and Signal Processing, pp. 212-228, 1988.

Chapter 5

Example Fringe Analysis Applications

5.1 Introduction

This chapter describes several examples of the application of fringe analysis techniques, particularly the Minimum Spanning Tree approach to phase unwrapping (described in detail in Chapter Three). Examples of both Phase Stepping and FFT methods are given.

5.2 Deformation Measurement of a Metal Disc

In this example, the practical work of making the hologram, the use of fibre optics to create the carrier fringes and the theoretical discussion of disc deformation have been conducted by Chenggen Quan, Engineering Department Warwick university. The analysis phase, image capture software, FFT processing, correction for non-linearity of the carrier and phase unwrapping have been conducted by the author. This collaboration is a good illustration of the multidisciplinary nature of the subject. The example is the subject of a paper to be published in Optical Engineering [1].

A carrier fringe technique for measuring surface deformation is described and verified by experiments. In contrast to conventional holography and fringe analysis, this holographic system is based on fibre optics and automatic carrier fringe analysis techniques. Single-mode optic fibres are used

to transfer both the object and reference beams. Carrier fringes are generated by simply translating the object beam between two exposures. The Fast Fourier Transform (FFT) method is used to process the interferograms. The experiment gives an example of the tile level minimum spanning tree phase unwrapping technique and the pixel level, noise immune, unwrapping strategy. The test object is a centrally loaded disc. An excellent correlation between the theoretical deformation profile and that suggested by the technique is given.

5.3 Introduction to Disc Deformation

Holographic interferometry is a powerful technique for measuring deformation, deflection and vibration. Unlike conventional interferometry, holographic interferometry can be used to make measurements on Lambertian surfaces. The hologram permits a three dimensional surface to be recorded, and investigated from different view points.

There are several ways of generating carrier fringes;

- i) By applying a very small tilt to the wavefront illuminating the object field between exposures [2].
- ii) By rotating a mirror so that the point source appears to have moved [3].
- iii) By tilting the object between two exposures [4].
- iv) By translating the illumination lens (in shearography strain measurement) [5].

Optical fibres offer advantages in the field of optical holography [6, 7, 8];

- i) They require much less space than a conventional system and allow almost unlimited freedom in selecting object and reference beam angles.
- ii) The fibre holographic system is very flexible, enabling interferometric studies of obscured or remote objects, or investigations in difficult environments.

5.4 Theory

5.4.1 Direct Deformation Measurement of a Disc

In order to demonstrate the fringe analysis method for measuring deformation, an aluminium disc has been chosen as an experimental object. A direct deformation measurement for an ideal clamped circular disc may be calculated from theory. The deflection of the disc at a distance r from the centre is given by [9]

$$w = \frac{Fr^2}{8\pi D} \ln \frac{r}{R} + \frac{F(R^2 - r^2)}{16\pi D} \quad (5.1)$$

where

$$D = \frac{Eh^3}{12(1 - \nu^2)} \quad (5.2)$$

Equations 5.1 and 5.2 are for a disc of radius R , thickness h , Young's modulus E and Poisson's ratio ν . The load, F , is determined from the change in deflection of the centre of the disc imposed between exposures. In the case of the aluminium disc, the parameters for Equations 5.1 and 5.2 are as follows, $R = 50\text{mm}$, $r = 20\text{mm}$, $E = 70\text{kN/mm}^2$, $\nu = 0.33$, $h = 2\text{mm}$. The deformation w_1 measured at the radial distance r was found to be $1.3 \mu\text{m} \pm 20\text{nm}$ via a precision gauge. Therefore, the force F may be calculated from the equations above as

$$F = \frac{w_1}{\frac{r^2}{8\pi D} \ln \frac{r}{R} + \frac{R^2 - r^2}{16\pi D}} = 2.5 \text{ N} \quad (5.3)$$

From Equations 5.1 and 5.2, the maximum deflection at the centre of the clamped disc, when loaded at the centre, is given by

$$w_{\max} = \frac{FR^2}{16\pi D} = 2.38 \mu\text{m} \quad (5.4)$$

5.4.2 Objective of Holographic Carrier Fringe Technique

The deformation fringe pattern obtained from double-exposure holographic interferometry can be complex. This complexity can be reduced to yield

a simple fringe pattern using carrier fringe linearisation. This is the first objective of holographic carrier fringe techniques. The second objective is to yield a single interferogram for analysis, which encodes direction as well as displacement. In this carrier fringe technique, the fringes are introduced in the formation step of a double-exposure hologram by shifting the fibre optic beam, illuminating the object, between the two exposures. This principle of shifting the object wavefront between exposures to create linear cosine fringes has been illustrated in [10]. The fringes have a period ρ in object space, which is given by [6]

$$\rho = \frac{\lambda}{\sin \frac{\gamma}{2}} \quad (5.5)$$

where λ is the wavelength of the laser light and γ is the amount of angular shift between the object beams.

5.5 Image Processing Technique

5.5.1 Windowing to Isolate a Side Lobe

Each raster of the interferogram produces a power spectrum similar to that shown in Figure 5.9. Each of the spectral side lobes represent modulated fringes contouring the measurement parameter. The centre lobe represents unwanted background variations in the intensity of the interferogram, which are low in frequency.

The FFT technique eliminates background variations by considering just one of the side lobes. The frequency tails of the lobes may overlap, if the carrier frequency is insufficient. An appropriate division point between the lobes must be found. An upper frequency limit for the side lobe must also be found. Once these points have been determined, a window may be used to isolate the side lobe, using the cut points to determine the size of the window.

A variety of strategies may be employed to decide upon these cut points. Here, the average power in a band of frequencies (the width of the check band has been specified as a third of the carrier frequency) to either side of the carrier has been measured at successive offsets from the carrier. The criterion for these cut points is that the average power in the check bands must fall

below a certain threshold, on both sides of the carrier. This threshold is initially set very low. If no bands are found with an average power less than the threshold, then the threshold is incremented slightly, and the process repeated. The window function is computed over the range which this test suggests, the window may differ, from scan line to scan line.

A Papoulis window has been employed in the spatial domain to bring the spatial data to zero at the edges of the image. This window has been selected for its sharp cut off. It is obtained by square rooting the Hanning window. A Hanning window has been used in the frequency domain to isolate the side lobe. These windowing procedures can have dramatic effects on the quality of the results obtained.

5.6 Experimental Description and Results

5.6.1 Experimental Set-up

A schematic diagram of the experimental system is shown in Figure 5.1. The object is a centrally loaded aluminium disc which has a 100 mm diameter. The light from a 25mW He-Ne laser ($\lambda = 0.6328\mu\text{m}$) was launched into the expander and divided in two by a (60/40) beam splitter (BS). Two single-mode optical fibres of about 2m in length were used, one arranged to illuminate the object and the other arranged to illuminate the holographic plate (H). The object is recorded before and after the deformation by the double-exposure technique. As mentioned earlier the deformation was measured by a precision gauge, which has an accuracy of $\pm 20\text{nm}$. In order to generate carrier fringes for quantitative analysis, the object beam was moved between the exposures by a micro translator. The number of carrier fringes is determined by the amount of movement of the object beam.

The carrier fringes should be of high enough density to clearly distinguish the side lobe from the centre lobe. In this case, the frequencies of the deformed fringes are small compared with the spatial carrier. In this experiment the carrier frequency was slightly lower than the ideal. That is, the side lobe is not clearly separated from the central lobe. The effect of this can be seen in the contour plot of this data shown in Figure 3.31 (Chapter 3). There

is a slight ripple in the contours above and below the central peak. If these areas are inspected in the original interferogram of Figure 5.5, it can be seen that these areas have respectively low and high frequencies when compared to the carrier. These frequencies, therefore, lie in the frequency tails to left and right of the side lobe in the power spectrum, see Figure 5.9. They are therefore particularly sensitive to defects in the windowing procedure which extracts the side lobe.

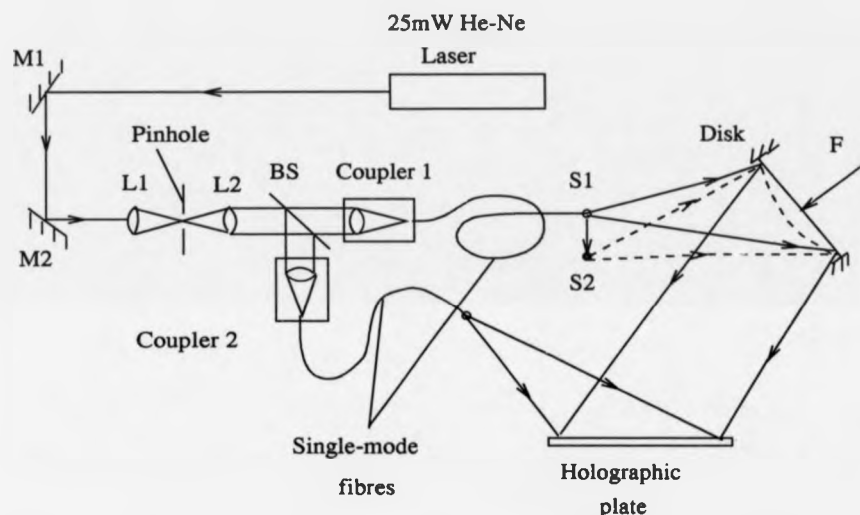


Figure 5.1: Experimental Arrangement for Recording Holograms of the Disc by the Carrier Fringe Technique

An exposure was taken with the illuminating object beam at position $S1$ in Fig 5.1. Then the object beam was translated $200\text{ }\mu\text{m}$ sideways to position $S2$ to form the carrier fringes, after which a second exposure was made on the same plate. Between the exposures, the disc was deformed by central loading. The deflection at the centre was estimated from the calculated force as $2.38\text{ }\mu\text{m}$.

Figure 5.2 shows the system for reconstructing holograms. A reconstructed image is picked up by a CCD camera and digitised as a 512×512 pixel image with 8 bits of intensity resolution. An IBM compatible micro-computer has served as host for the frame capture board. This is connected

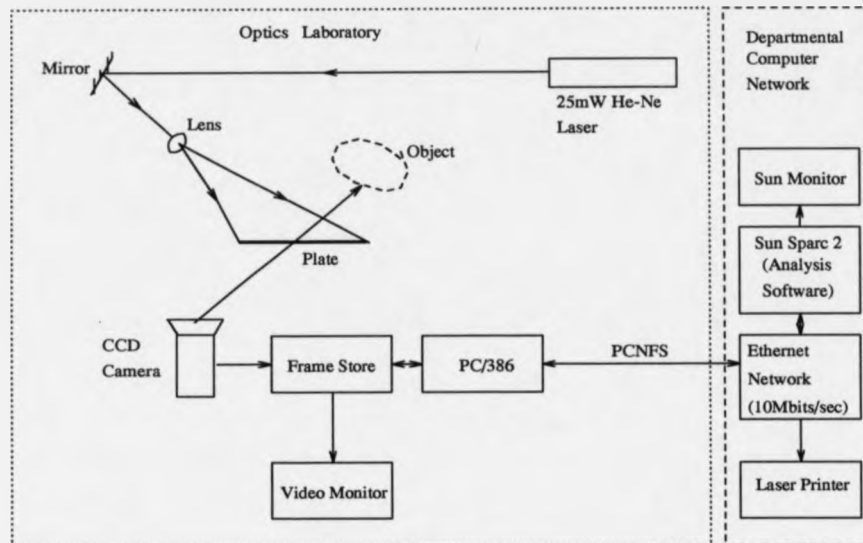


Figure 5.2: Optical and Electronic Arrangement for Reconstruction of the Holograms

via Ethernet and the PC Net Filing System (PCNFS) to the departmental Sun Network, (data transfer is transparent to the PC, which simply saves an image file to disc). Image processing takes place on a Sun Sparc station 2. The result may be read by the PC and displayed (again transparently by reading a file from disc). Hardcopy of the results is produced by a laser printer. See Figure 5.2.

5.6.2 Correction for the Non-linearity of the Carrier Fringes

The carrier fringe maxima, as seen around the edge of the deformed disc, can be shown to follow Equation 5.6;

$$x = \frac{m\lambda a}{b + m\lambda \tan \theta} \quad (5.6)$$

where ($m = 0, 1, 2, 3, \dots$) is the carrier fringe order number, x is the distance of the intensity maxima in the carrier fringes from an origin position

on the flat disc, b is the displacement between source position $S1$ and $S2$, and a is the distance from source to object. These quantities are illustrated in Figure 5.3.

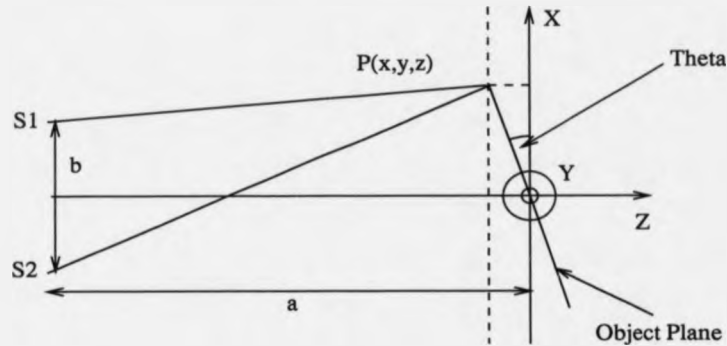


Figure 5.3: Schematic Diagram for Generation of Carrier Fringes at Slight Inclination

The effect of the non-uniformity in the carrier may be observed in the unwrapped data, see Figure 5.11. The difference between the function represented by Equation 5.6 and the ramp expected has been superposed on the unwrapped data.

In this analysis the carrier wavelength, employed to demodulate the deformation signal, has been selected as the carrier fringe spacing at the centre of the interferogram. Therefore, the solution is correct at the centre. However, the carrier wavelength, following Equation 5.6, becomes stretched, non-linearly, towards the top of the interferogram and compressed towards the bottom. This can be seen as the bowl shape in the unwrapped data of Figure 5.11. The bowl shape s is proportional to Equation 5.7.

$$s \propto \frac{m\lambda a}{b + m\lambda \tan \theta} - m\kappa \quad (5.7)$$

where κ is the carrier fringe spacing at the centre of the object.

This superposed deformation has been corrected for. This has been accomplished by fitting a polynomial to the data which represents the surround, at the edge of the interferogram, which did not undergo deformation. The profile of this fitted curve has then been subtracted across the unwrapped

map.

In addition to the above it is noted that the carrier fringes are also not exactly aligned with the axes of the frame. A slight ramp running across the image has, therefore, also been imposed. This has been corrected for, in a similar fashion. If a 2D FFT analysis had been applied, then this problem could have been identified in the Fourier domain and compensated for.

5.6.3 Results and Evaluation

The interferogram obtained by making a double exposure hologram of the disc before and after applying the central loading is shown in Figure 5.4. The interference fringes are basically a set of concentric rings. Figure 5.5 shows the fringe pattern produced by both carrier and deformation present during the recording. The carrier has been created so that the fringe orders increase monotonically from top to bottom. The location of fringes in the undeformed carrier can be seen around the edge of the disc.

Figure 5.6 shows the wrapped phase map produced by the FFT process. Figure 3.30 shows the edge detection of the wrapped phase map, using a Sobel with 3×3 kernel and adaptive thresholding. Figure 5.7 shows a normalised grey scale plot of the unwrapped phase before it has been corrected for the non linearity of the carrier fringes.

The input data for computer processing consists solely of the digitised image of the hologram. The functions and variables referred to in this section are defined in the discussion of the FFT technique in Chapter 2, Section 2.5. By applying the inverse FFT of $C(u, y)$ with respect to u , $c(x, y)$ is obtained. Figure 5.8 (a) shows the intensity distribution along raster 256 of the 512 in the image. The unwanted irradiance variations which are expressed by $a(x, y)$ and $b(x, y)$ can be seen in Figure 5.8 (a), for example between x-pixel indices 100 and 200. The end points of each raster are brought to zero by applying a Papoulis window across each raster in the spatial domain, to simulate a periodic function, as shown in Figure 5.8 (b). The Fourier transform of the data in Figure 5.8 (b) is shown in Figure 5.9 (a). Figure 5.9 (b) shows the side lobe translated by f_0 in Figure 5.9 (a) toward the origin to obtain $C(u, y)$.

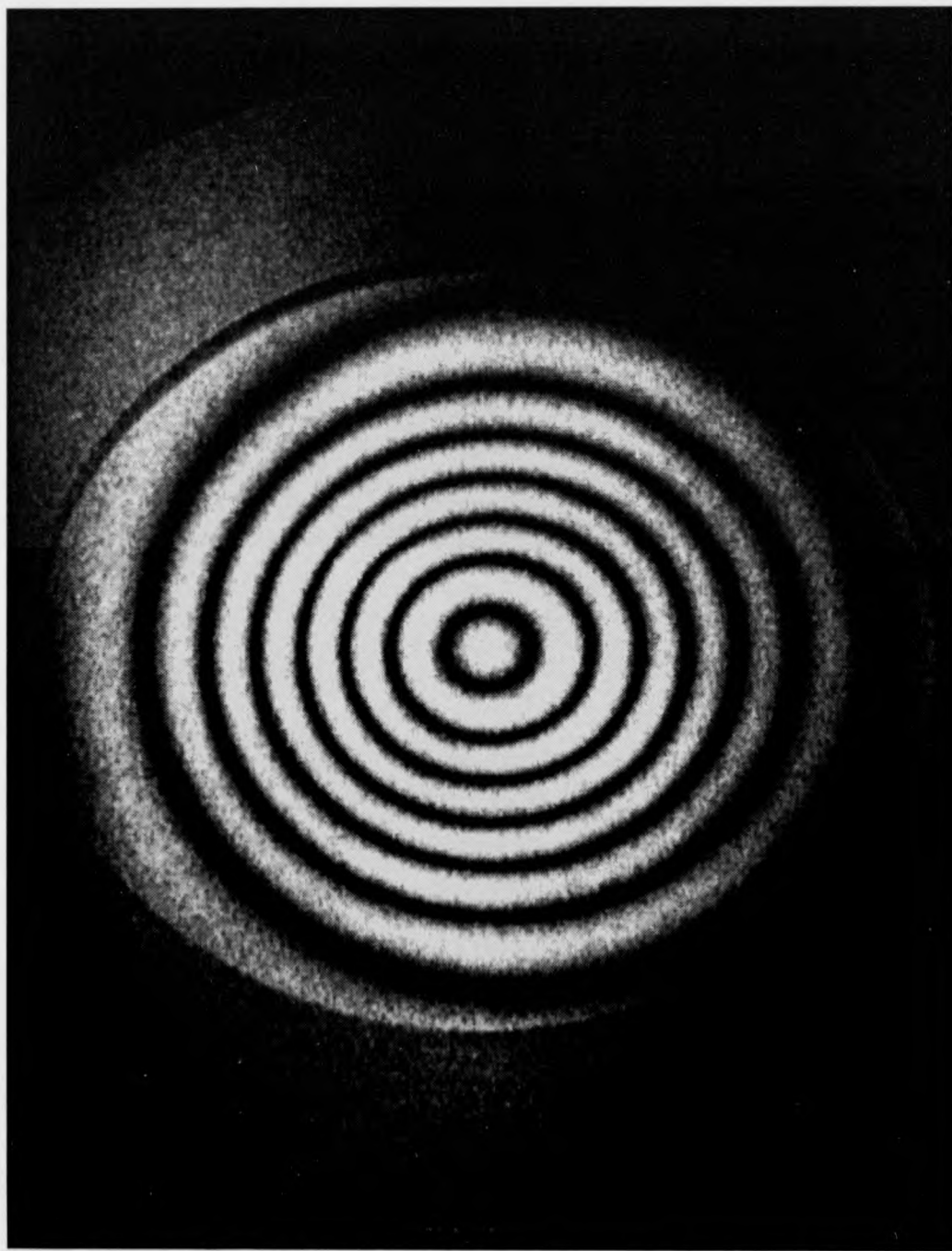


Figure 5.4: Double-exposure Holographic Interferogram of the Centrally Loaded Disc

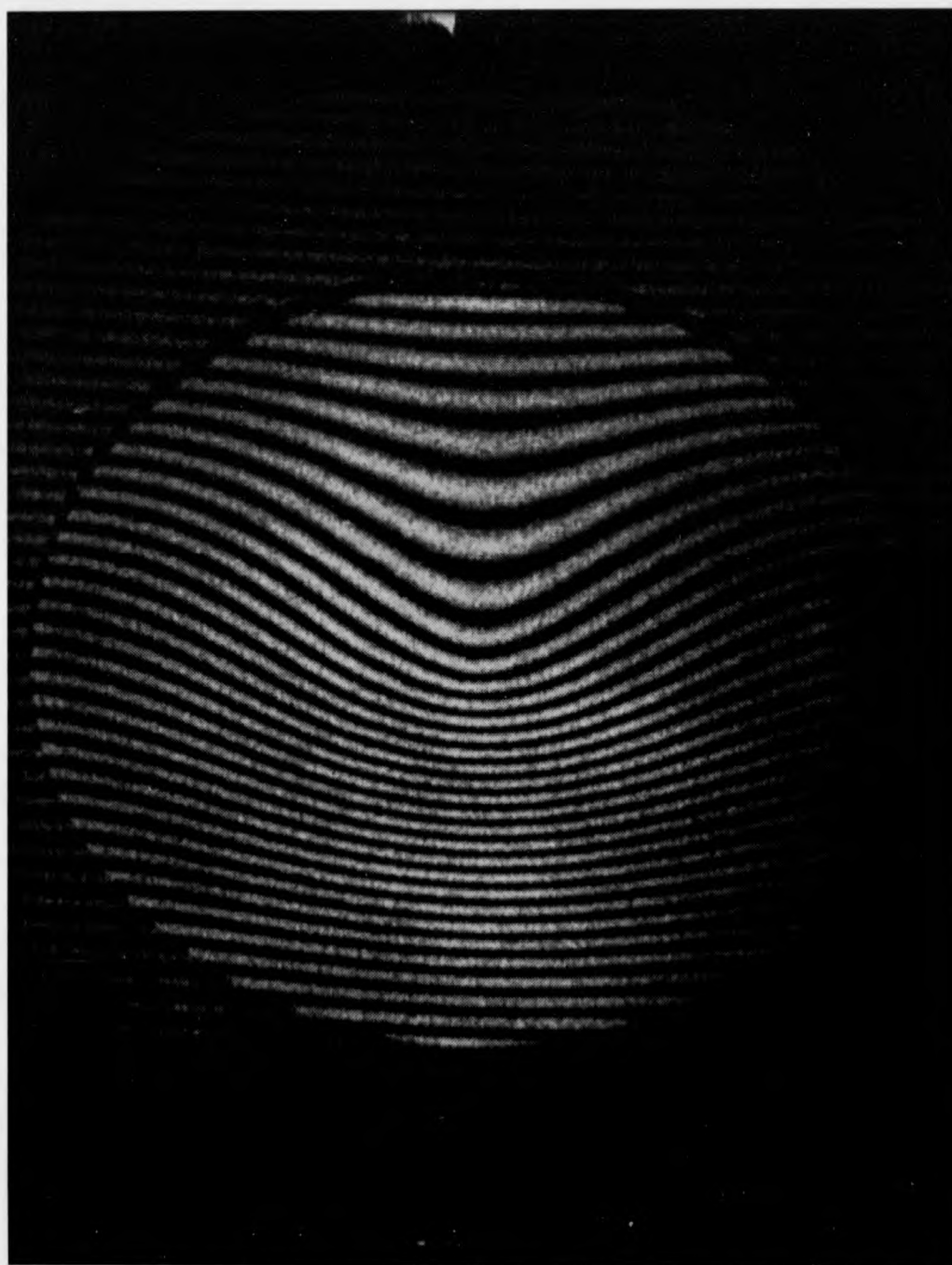


Figure 5.5: Holographic Carrier Fringe Interferogram of the Centrally Loaded Disc



Figure 5.6: Wrapped Phase Map for Centrally Loaded Disc, Generated by the FFT Technique



Figure 5.7: Normalised grey scale plot of deformation produced by unwrapping procedure, before correction for non linearity of carrier fringes.

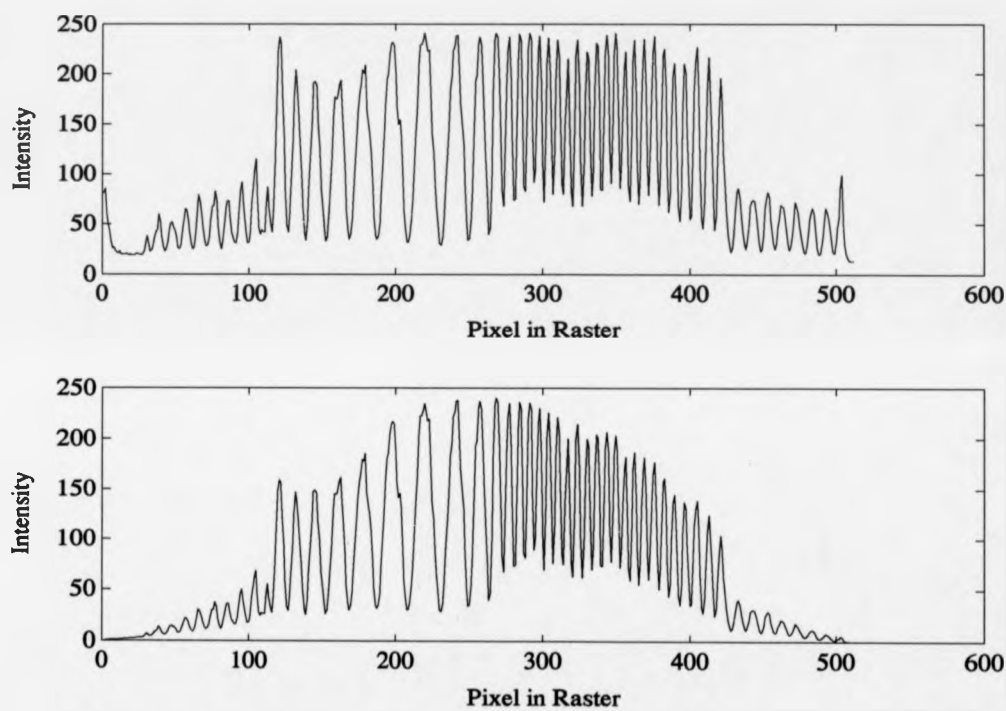


Figure 5.8: (a) Digitised intensity data of central raster in the interferogram; (b) intensity data weighted by Papoulis window.

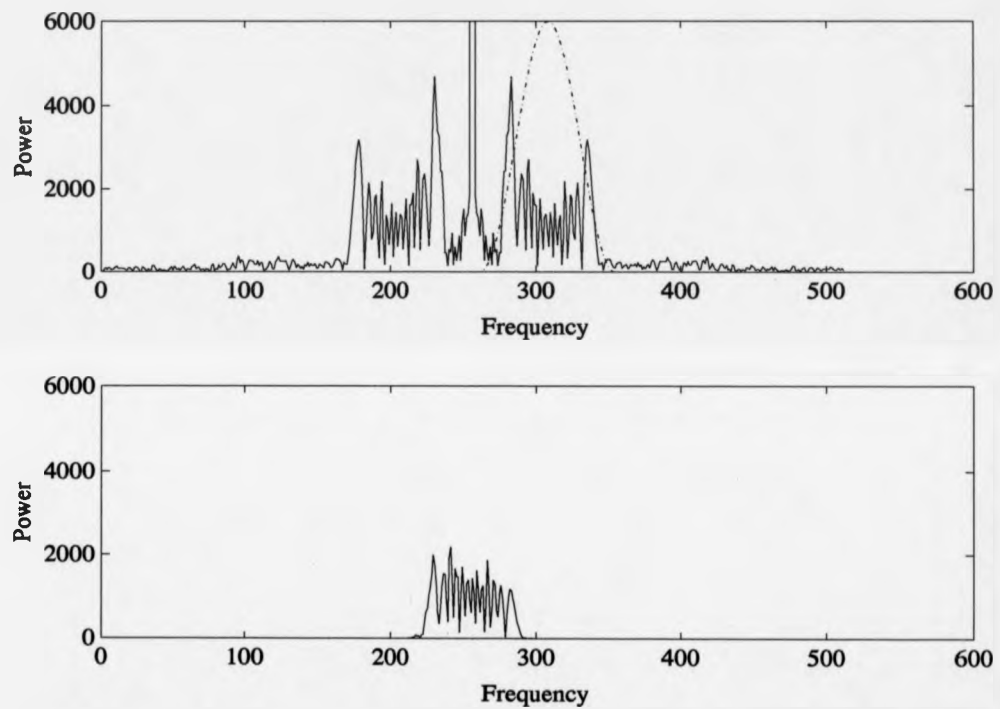


Figure 5.9: (a) Power spectrum of central raster from the interferogram with carrier and deformation (the window is indicated by the dashed dot line); (b) side lobe translated by the carrier frequency to the origin position.

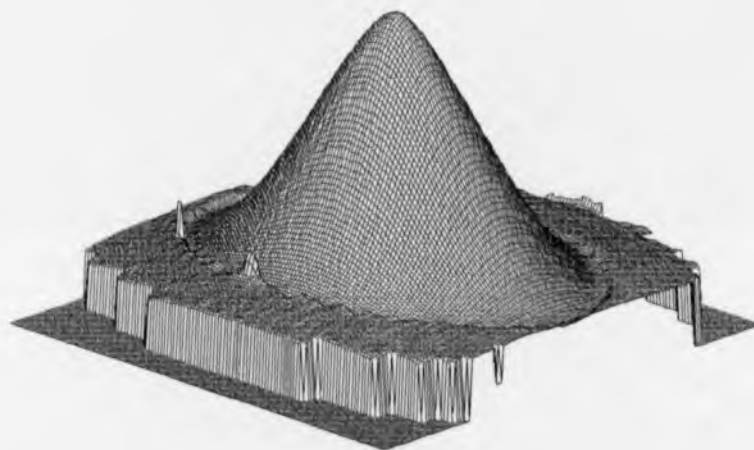


Figure 5.10: 3-D perspective plot of the out of plane displacement of the centrally loaded disc after phase unwrapping and correction.

The phase unwrapping algorithm counts up the agreements at tile boundaries across the interferogram to give an estimate of the field area solved, in the case of the disc with a tile size of 44 pixels including an overlap of 4 pixels.

- i) Total number of tile sides in complete frame = 624
- ii) Number of sides which computed between tiles = 480
- iii) Number of sides which did not appear to match = 2
- iv) Estimate of frame area solved = 76.60%

The 3-D perspective plot shown in Figure 5.10 shows the shape of the deformed disc after deformation. A side view of the unwrapped phase is shown in Figure 5.11, using the carrier frequency appropriate for the centre of each raster as stated above. The same view is shown in Figure 5.12 after the phase values have been corrected for the non uniformity of the carrier.

The experiment was so arranged that there existed a large separation between the disc and the fibre optic set up. In this case the theoretical out-of-plane displacement component w_{th} is given by [4]

$$w_{th} = \frac{m\lambda}{2\cos\alpha} \quad (5.8)$$

where m is the fringe number, λ the wavelength and 2α the angle between the propagation vectors in the direction of illumination and observation.

In fact Equation 5.8 is only valid for symmetric cases where the illumination angle and the viewing angle are equal valued on either side of the surface normal. If this condition were not followed an error would be introduced. The rate of change of the out-of-plane displacement, with respect to deviation of the illumination angle from its symmetric position is given by

$$\frac{dw_{th}}{d\alpha} = -\frac{m\lambda}{2} \cot\alpha \quad (5.9)$$

assuming that the observation angle remains unchanged.

By moving a cursor over the unwrapped map, the displacement between two selected points could be calculated. Single precision floating point has been employed in all computations following data capture.

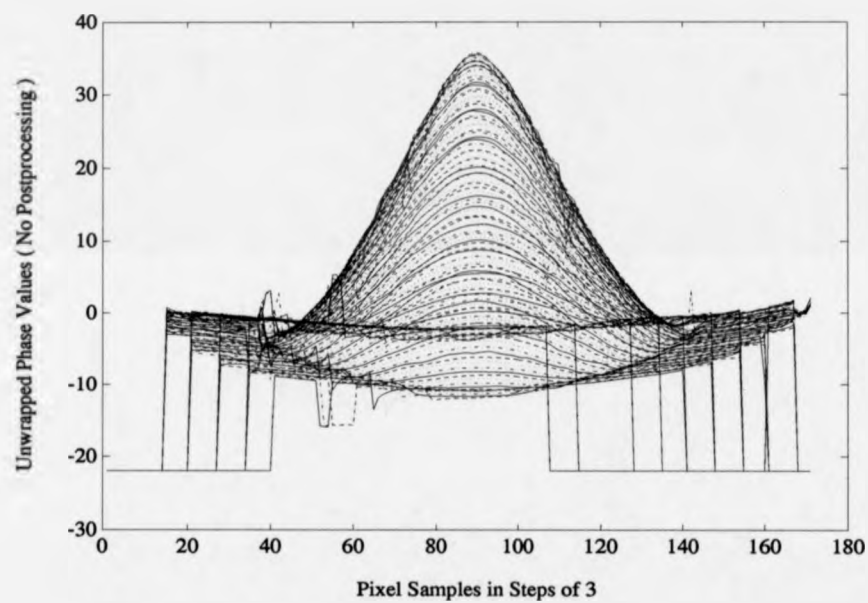


Figure 5.11: Side view of unwrapped numerical phase data.

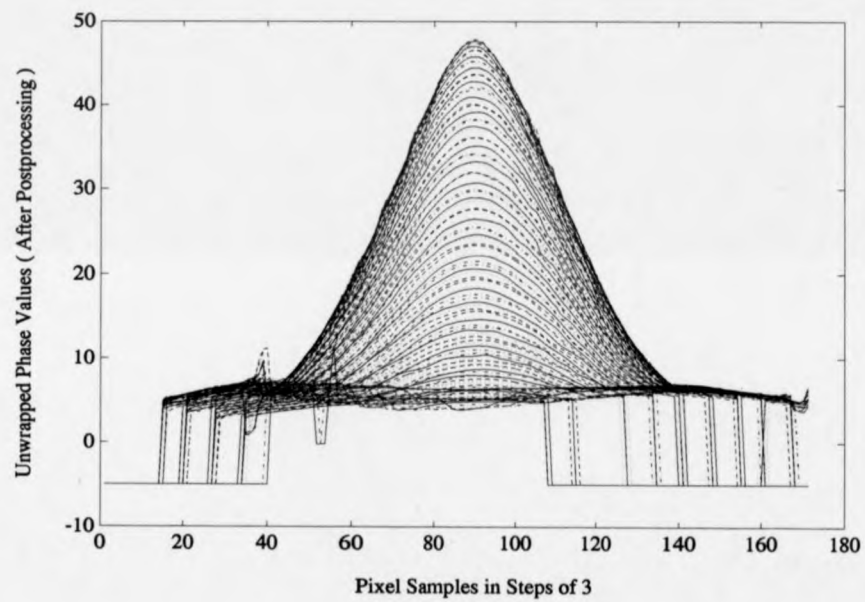


Figure 5.12: Side view of the unwrapped and corrected numerical phase data.

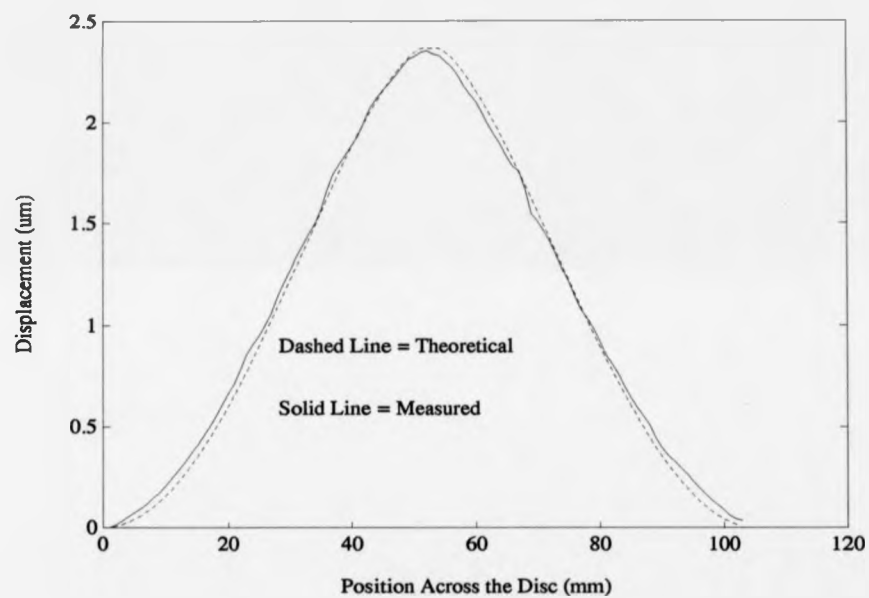


Figure 5.13: Comparison between the theoretical and measured deformation for cross section.

A point on the disc edge and a second at the maximum were sampled. By applying Equation 5.8 to the unwrapped phase values of these points the relative deformation was found. The angle α was measured as 25° . The maximum displacement at the centre was calculated as $2.35 \mu\text{m}$ in this way. The discrepancy between the theoretical deformation and that measured is therefore $0.03 \mu\text{m}$.

A cross section through the point of maximum deformation, obtained from the phase unwrapping procedure, is shown in Figure 5.13, along with a plot of the theoretical deformation curve. From this comparison it is seen that the maximum deviation in the comparison occurs at the edges of the disc. This result may be partially explained by the imperfect clamping of the disc at its edges, which means that a larger than expected deformation might appear at the disc edges.

5.7 Conclusion of Disc Deformation Measurement

The spatial carrier fringe technique has been applied to holographic interferometry. The spatial carrier across the object can be easily generated by moving fibre optics which illuminate the object. The use of fibre optics to carry the object and reference beams in holographic interferometry greatly facilitates the carrier fringe technique and the FFT fringe analysis method. The non uniformity of the carrier fringes has been dealt with by considering the non uniformity as a superposed deformation.

It has been shown that the Fourier transform method of fringe analysis combined with the carrier fringe technique can be used to extract deformation information automatically from complex interferograms.

The image processing phase, including the FFTs, automatic window selection, generation of wrapped phase map, phase unwrapping, and storage of grey scale image and numeric ascii data (on 3 pixel grid) were all executed in software and required 1 minute 15 seconds of processing on a Sun Sparc station 2.

5.8 Holographic Flow Visualisation

Carren Holden and Steve Parker of British Aerospace have been using the FRAN fringe analysis package developed in the course of this work to unwrap some of their holographic interferograms.

The test case presented below is a finite fringe hologram of a NACA 0012 aerofoil at a freestream Mach Number of 0.8. The Hologram was taken at the Cranfield Institute 9" wind-tunnel by the Sowerby Research Centre of British Aerospace PLC (Image published by kind permission). The unwrapped image represents a refractive index field, which is proportional to flow density.

Fringe analysis Work by Carren Holden and Steve Parker has been centred around the development of a 2D FFT analysis procedure. This procedure is being developed to take account of the additive and distorting effect of the Fourier transform of the model around which the flow is moving. The model appears as solid, without fringes in Figure 5.14. The analysis procedure does not window in the frequency domain, as was the case in the disc example described in the previous section. A comparison is given between the automatic windowing procedure implemented within FRAN, used to analyse the disc, and the subtractive technique developed at BAe. The wrapped phase map computed by BAe has been processed so that the pixels in the area of the model are labelled as bad data points. By contrast, the detection of the model by the thresholding strategy implemented within FRAN is on a per tile basis.

The tile size used for both sets of images was 20 pixels with an overlap of 4 pixels.

Figure 5.14 shows the original image with carrier fringes. The windowed sequence is given first via the 1D FFT method. Figure 5.15 shows the wrapped phase map. As this is a one dimensional analysis, a slight ramp across the image remains. This can be seen by comparing Figure 5.15 with the wrapped phase map produced by the 2D analysis in Figure 5.19. Figure 5.16 shows the edge detection of the wrapped phase map and the tree followed during the phase unwrapping procedures. Figure 5.17 shows a grey scale image representing the unwrapped phase solution.



Figure 5.14: Finite Fringe Hologram of a NACA 0012 Aerofoil at a Freestream Mach Number of 0.8, copyright British Aerospace PLC 1991 for publication



Figure 5.15: Wrapped Phase Map by 1D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication

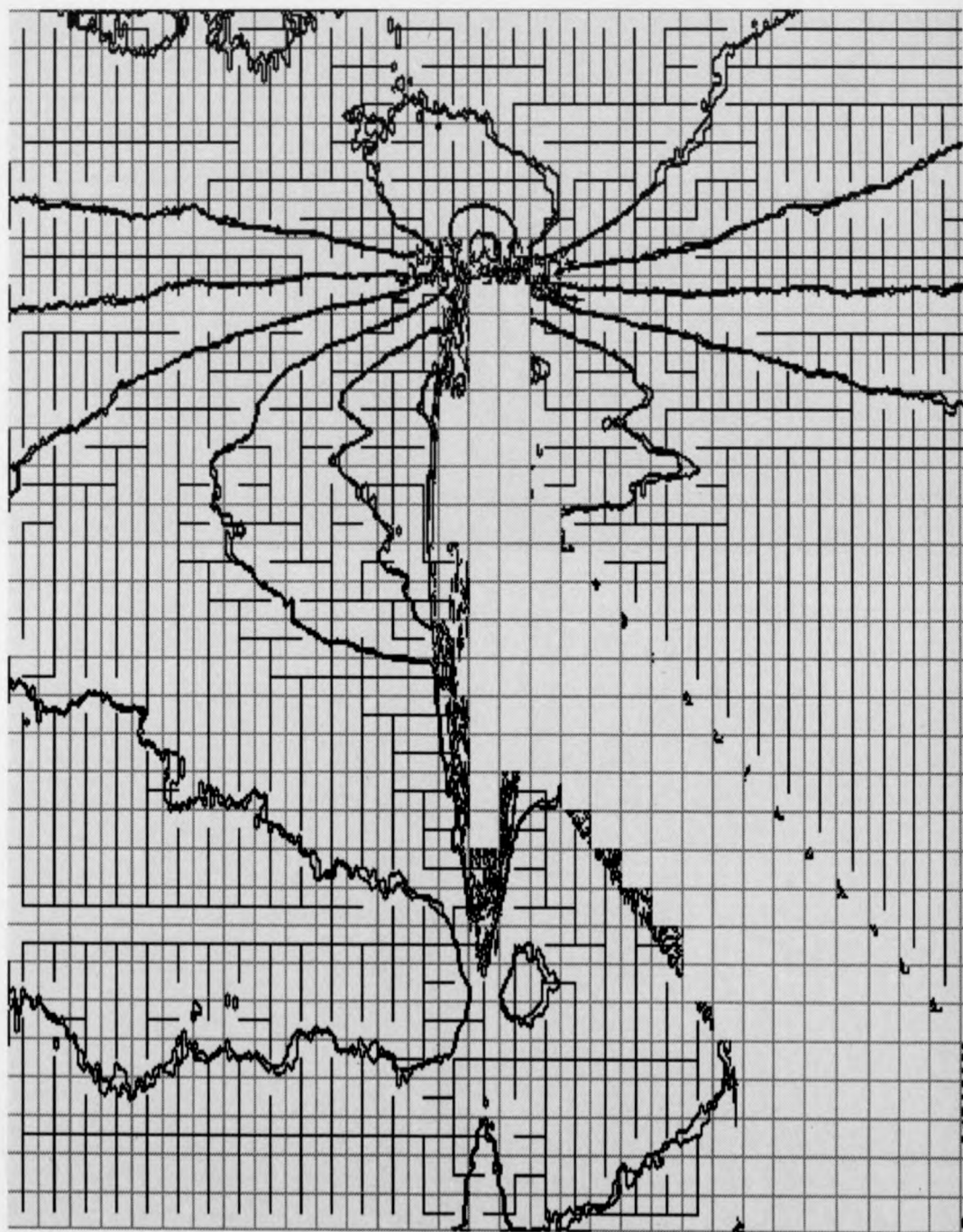


Figure 5.16: Edge Detection and Tile Connection Tree via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication



Figure 5.17: Grey Scale Unwrapped Phase Map by 1D FFT Method via
FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for
publication

The unwrapping process gave the following statistics for the area of the field which was solved.

- i) Total number of tile sides in complete frame = 3968
- ii) Number of sides which computed between tiles = 3284
- iii) Number of sides which did not appear to match = 62
- iv) Estimate of frame area solved = 81.20%

There then follows the sequence of images generated by Carren and Steve. Figure 5.18 is basically an averaged background image computed from Figure 5.14. The normalised 2D Fourier transform of this image is subtracted from the normalised 2D transform of Figure 5.14 to substitute for the Fourier windowing operation.

Figure 5.21 shows a contour map of the unwrapped phase solution for the 2D case. Figure 5.22 shows a mesh plot of the unwrapped phase for the 2D case.

The unwrapping process gave the following statistics for the area of the field which solved.

- i) Total number of tile sides in complete frame = 3968
- ii) Number of sides which computed between tiles = 3340
- iii) Number of sides which did not appear to match = 33
- iv) Estimate of frame area solved = 83.34%

5.9 Analysis of the Modes of Vibration of a Vibrating Board by Phase Stepping

This example was originally recorded in a dual reference beam holographic system. The example records the modes of vibration of a plane sheet caused to vibrate with an impinging air jet. The Holograms were produced using a double pulsed ruby laser. The experiment is fully described in reference [11].

The tile size for this image was specified as 24 pixels with a 4 pixel overlap.



Figure 5.18: Averaged Background Image for NACA 0012 Aerofoil, copyright
British Aerospace PLC 1991 for publication



Figure 5.19: Wrapped Phase Map by 2D FFT Method for NACA 0012 Aerofoil at a Freestream Mach, copyright British Aerospace PLC 1991 for publication

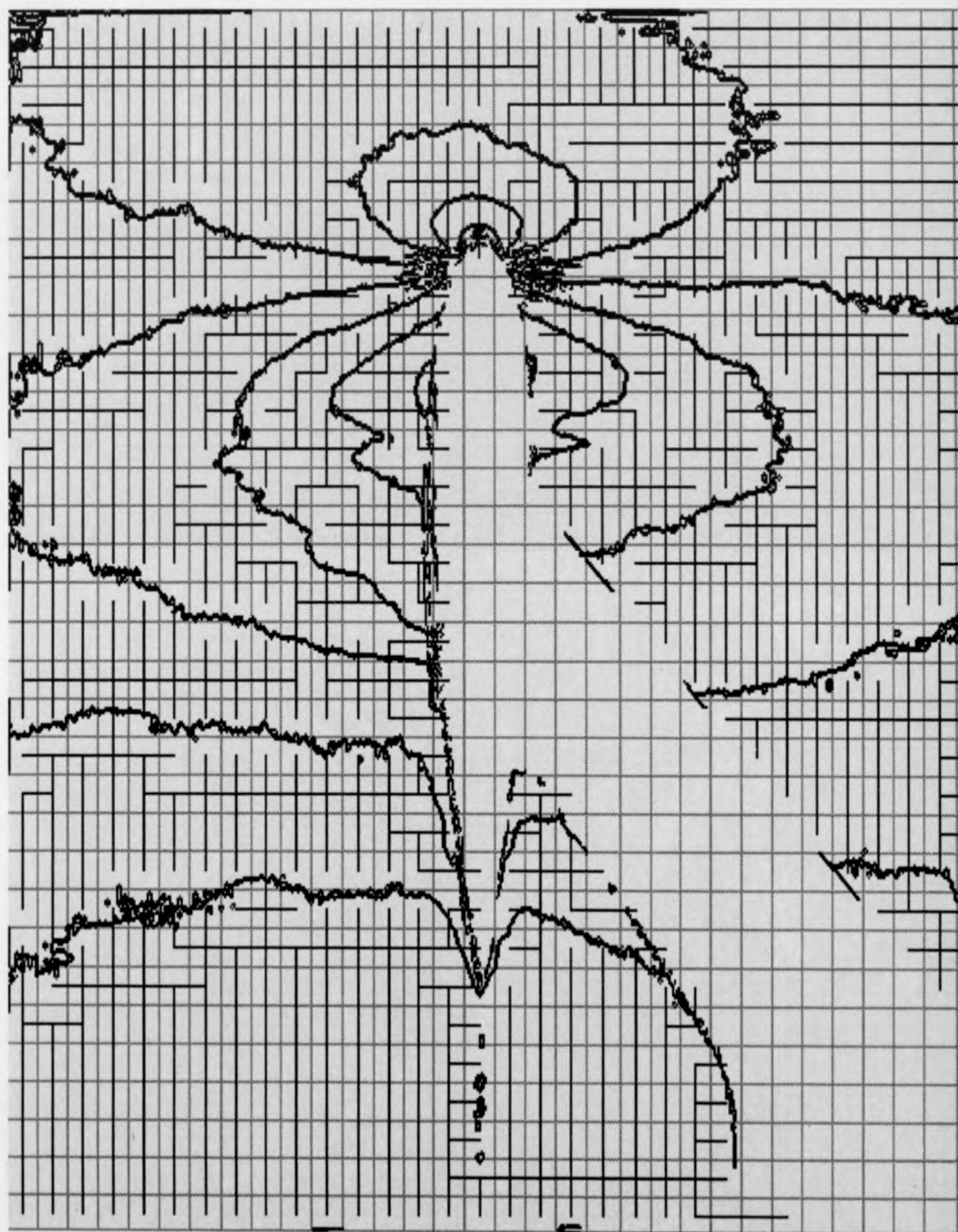


Figure 5.20: Edge Detection and Tile Connection Tree via FRAN for 2D FFT Analysis of NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication

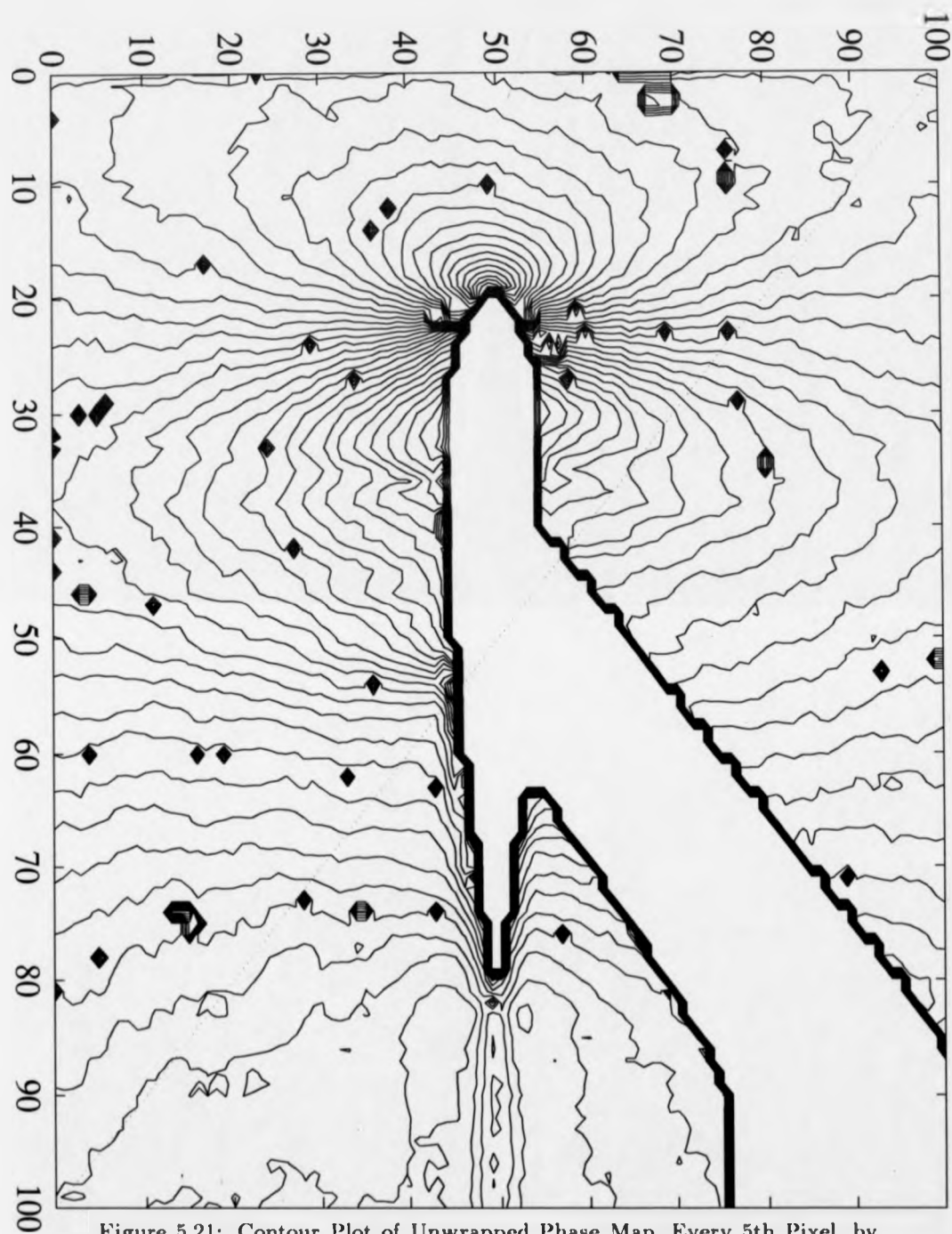


Figure 5.21: Contour Plot of Unwrapped Phase Map, Every 5th Pixel, by 2D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication

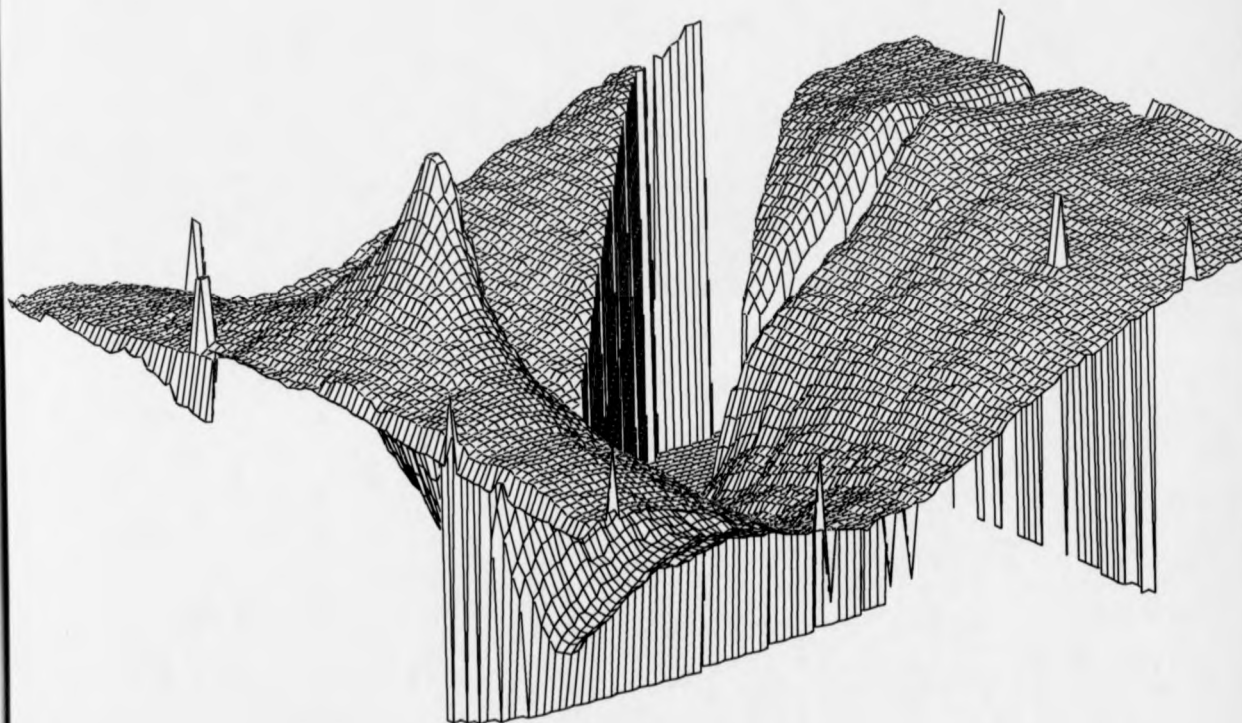


Figure 5.22: Mesh Plot of Unwrapped Phase Map, Every 5th Pixel, by 2D FFT Method via FRAN for NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication



Figure 5.23: Grey Scale Unwrapped Phase Map by 2D FFT Method via FRAN for Finite Fringe Hologram of a NACA 0012 Aerofoil, copyright British Aerospace PLC 1991 for publication 237

The 3 phase step method of analysis was applied with a phase step of 120 degrees between the images. Figure 5.24 shows one of the original interferograms, after prefiltering with a single iteration of a 3×3 averaging filter. Figure 5.25 shows the wrapped phase map for the vibrating board. Figure 5.26 shows the low modulation points that were detected. Figure 5.27 shows the edge detection and tile connection tree. Figure 5.28 shows the unwrapped solution as a grey scale image, where black is low, and white is high. Any tiles with more than 25% of their area consisting of low modulation points, have been rejected during the analysis.

This example is interesting because the impinging air jet has divided the field in two. Along the dividing line, aliasing has occurred. That is, the fringe density has exceeded the resolution of the capture system. It is instructive to explore the behaviour of the phase unwrapping algorithm in this case.

Examine the connection tree in Figure 5.27. It can be seen that information is missing along the line where the air jet has struck. The air jet is causing a steep valley where it strikes. The phase solution for the two halves have been connected on the far right of the board, at a distance from the impinging jet, where the displacement is least. Because of the relatively small displacement, the fringe density is less in this area and so aliasing has not occurred. It may be verified by reference to the wrapped phase map, Figure 5.25, that this is the only region of valid data connecting the two halves of the board. This area has been chosen by the algorithm for the following reasons

- i) There is good agreement between the tiles to either side of jet valley.
- ii) There are no fringe edge termination points in the connecting tile, by contrast with the other tiles along the valley.
- iii) There are few low modulation points.

These three factors have over ridden the last factor, that is apparent fringe density, which is high in the area, to select this position as the most reasonable connection point.

- i) Total number of tile sides in complete frame = 2600

- ii) Number of sides which computed between tiles = 2070
- iii) Number of sides which did not appear to match = 295
- iv) Estimate of frame area solved = 68.27%

5.10 Soldering Iron, Aliasing on An Object In The Field

The last example shows an interesting example of the technique avoiding the aliased carrier fringes on a soldering iron, during FFT processing. Figure 5.31 shows the original interferogram with carrier fringes. Figure 5.32 shows the wrapped phase map produced after demodulation. The side lobe corresponding to the wrapped phase map was isolated by a rectangular window, instead of the papoulis, in the Fourier domain. This is therefore a very noisy example of a wrapped phase map, as high frequency components have been left in. Figure 5.33 shows the edge detection of the wrapped phase map and tile connection tree. Figure 5.34 shows the grey scale solution. In this image the mismatch in the tiles in the area of the soldering iron can be seen at once, the manner by which the algorithm avoids the area, by comparing the tile solutions, is clear. Figure 5.35 shows a contour map of this solution, and Figure 5.36 shows a mesh plot of the solution.

The tile size used was 24 pixels including the 4 pixel overlap. An estimate for the field area solved is given by the package:

- i) Total number of tile sides in complete frame = 2600
- ii) Number of sides which computed between tiles = 1510
- iii) Number of sides which did not appear to match = 137
- iv) Estimate of frame area solved = 52.81 %



Figure 5.24: Original Holographic Cosinusoidal Interferogram of Vibrating Board after Prefiltering



Figure 5.25: Wrapped Fringe Field for Vibrating Board



Figure 5.26: Low Modulation Noise (in White) for Vibrating Board

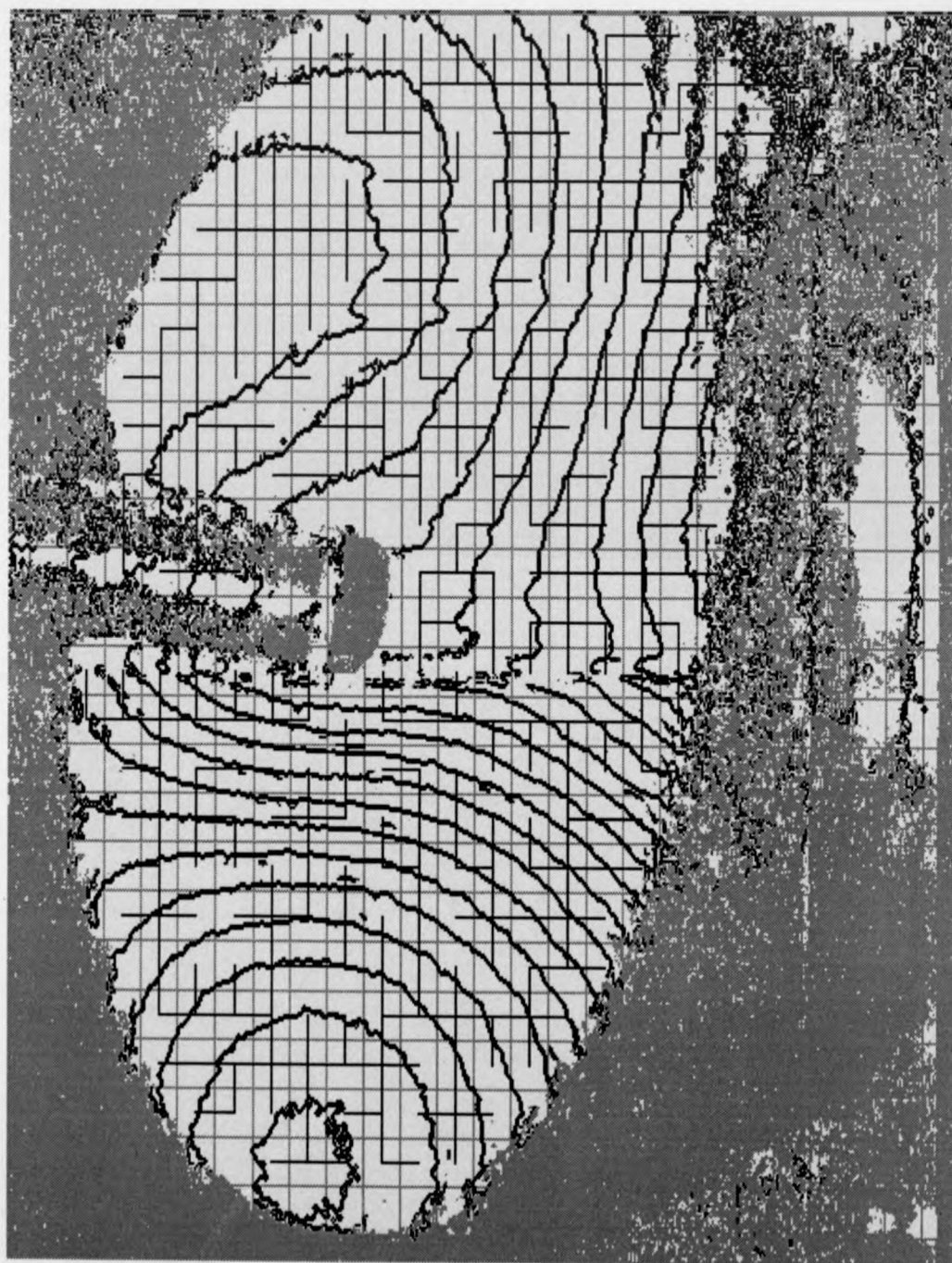


Figure 5.27: Edge Detection of Wrapped Fringe Field for Vibrating Board, Showing Tiles and Connection Tree



Figure 5.28: Grey Scale Unwrapped Fringe Field for Vibrating Board

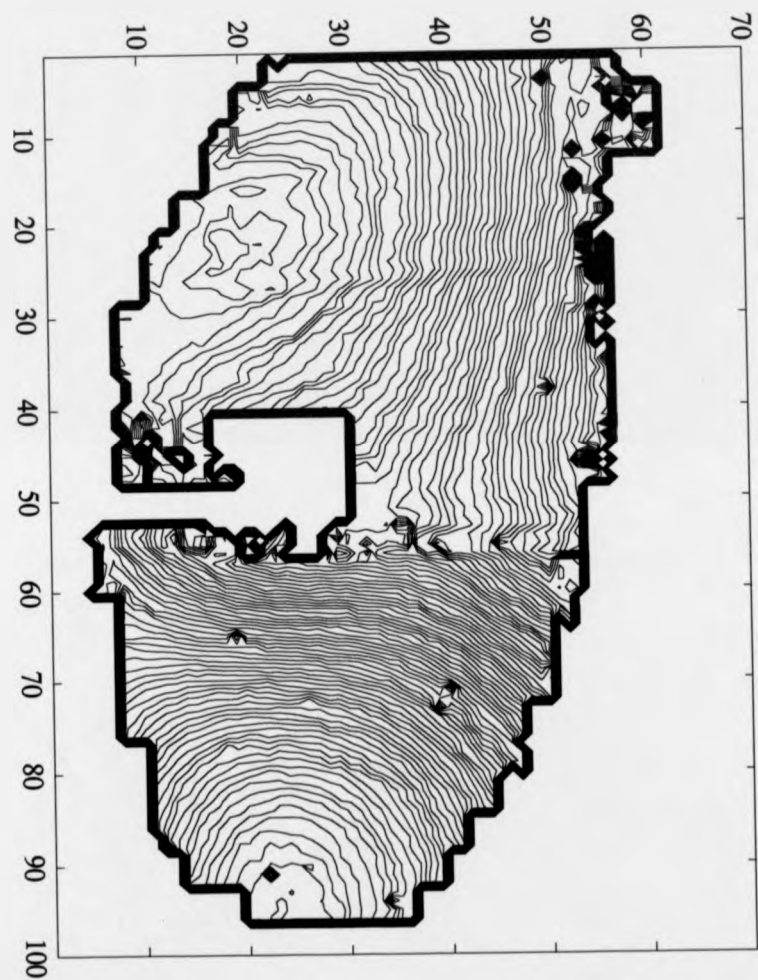


Figure 5.29: Contour Plot of Unwrapped Phase for Vibrating Board, Every 5th Pixel on Long Axis

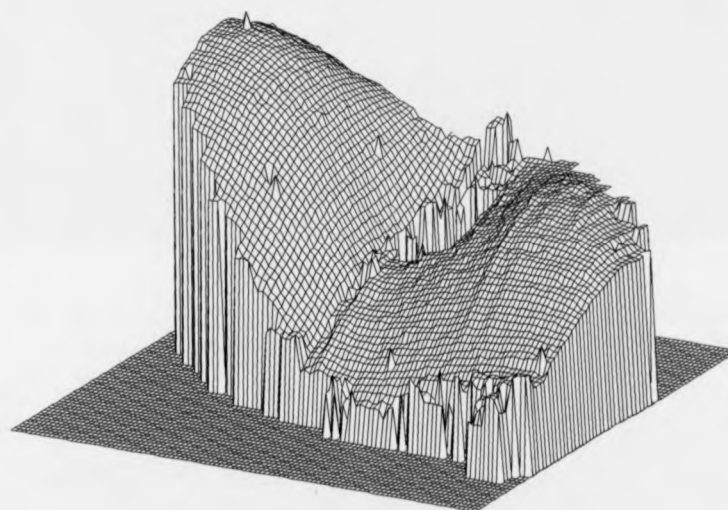


Figure 5.30: Mesh Plot of Unwrapped Phase for Vibrating Board



Figure 5.31: Soldering Iron Interferogram with Carrier Fringes

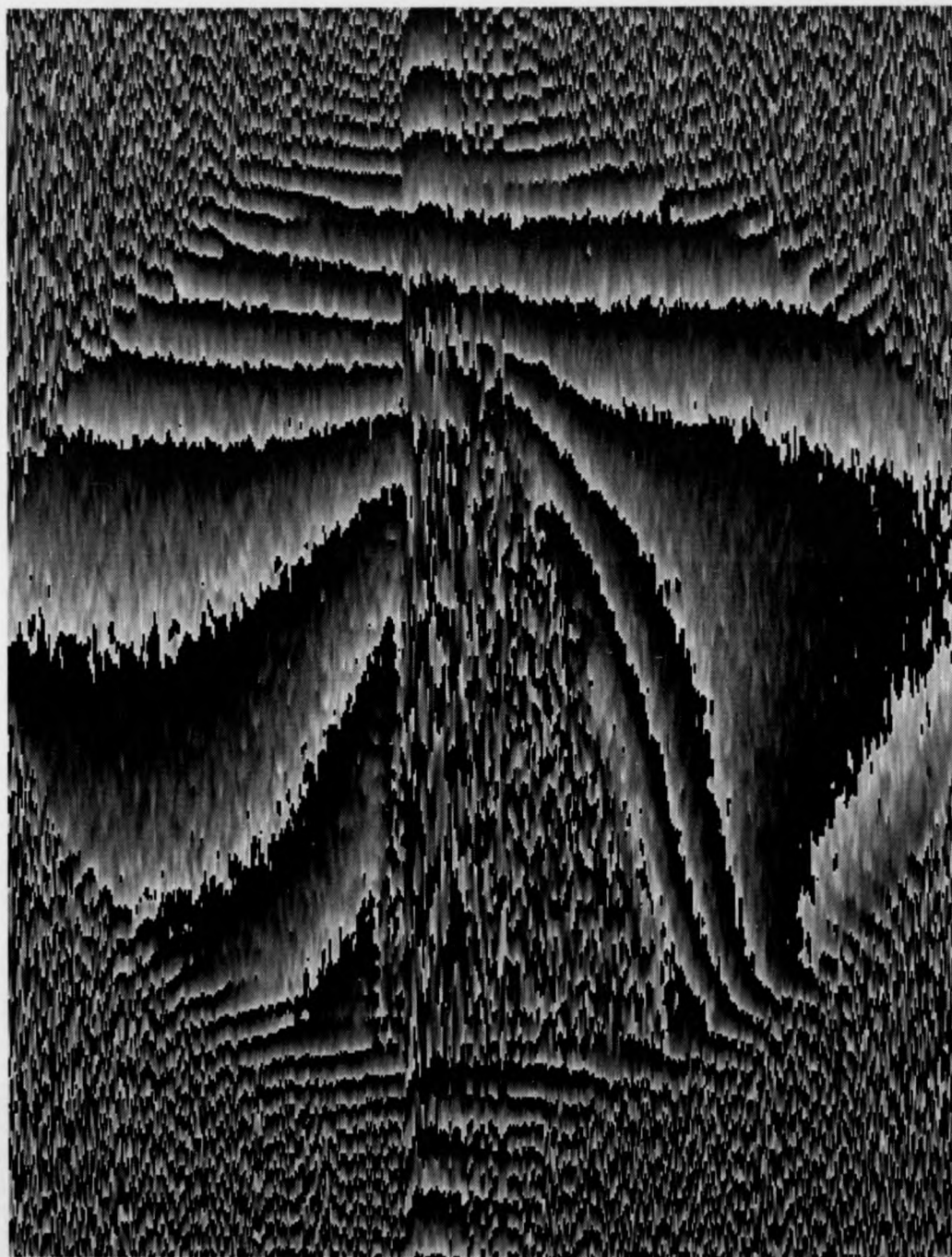


Figure 5.32: Wrapped Phase Map Of Soldering Iron

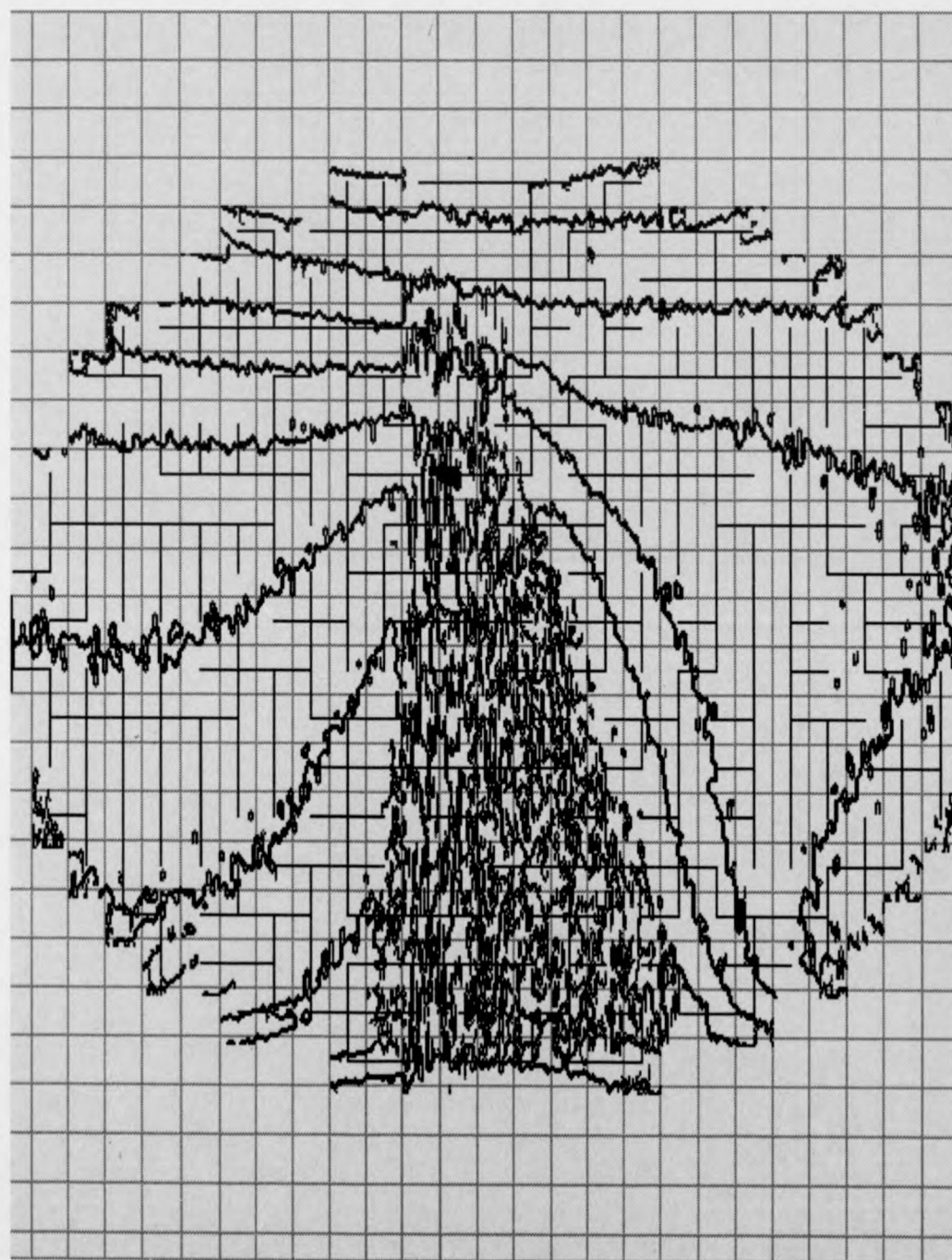


Figure 5.33: Edge Detection and Tile Connection Tree for Soldering Iron



Figure 5.34: Grey Scale Plot of Solution Showing Circumvention of Discontinuities in Area of Soldering Iron

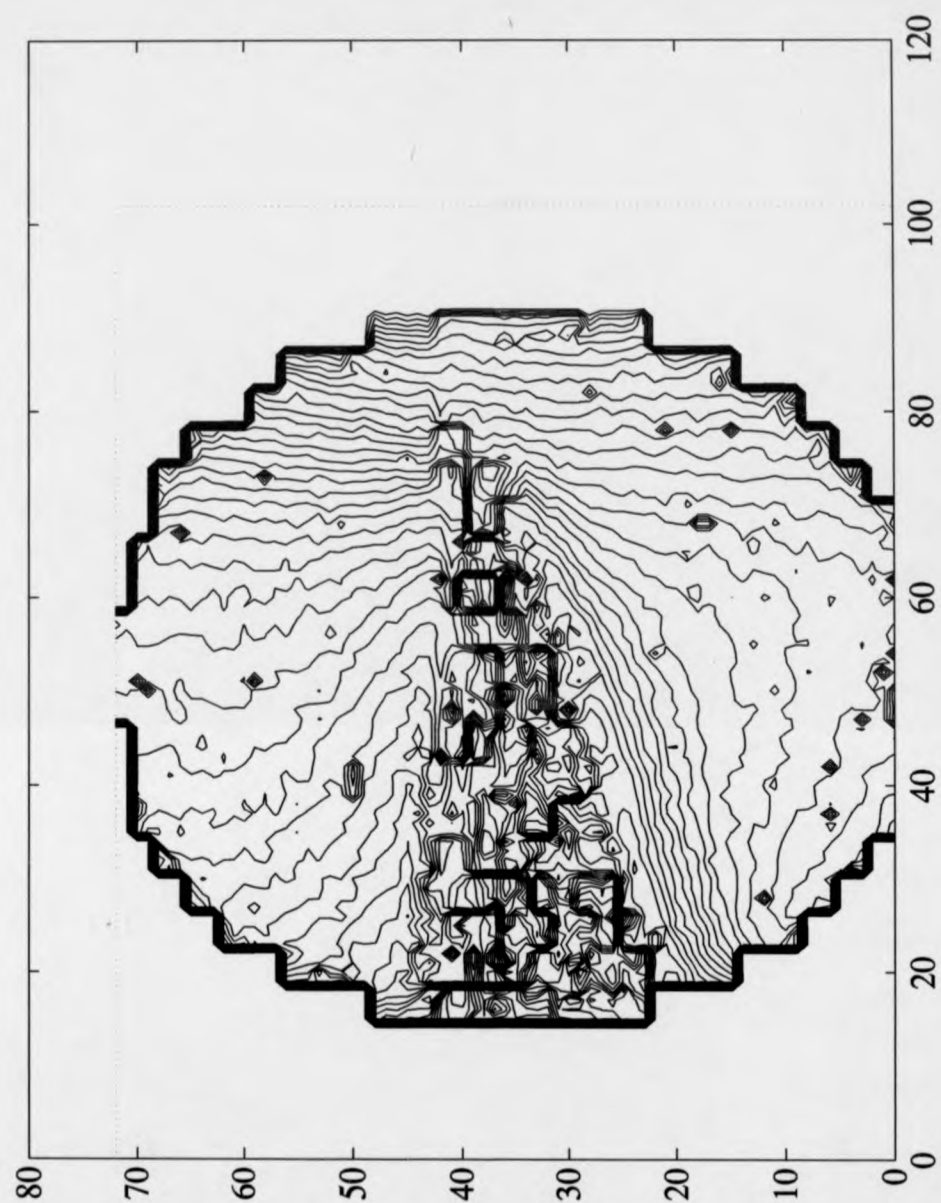


Figure 5.35: Contour Map Of Soldering Iron Solution

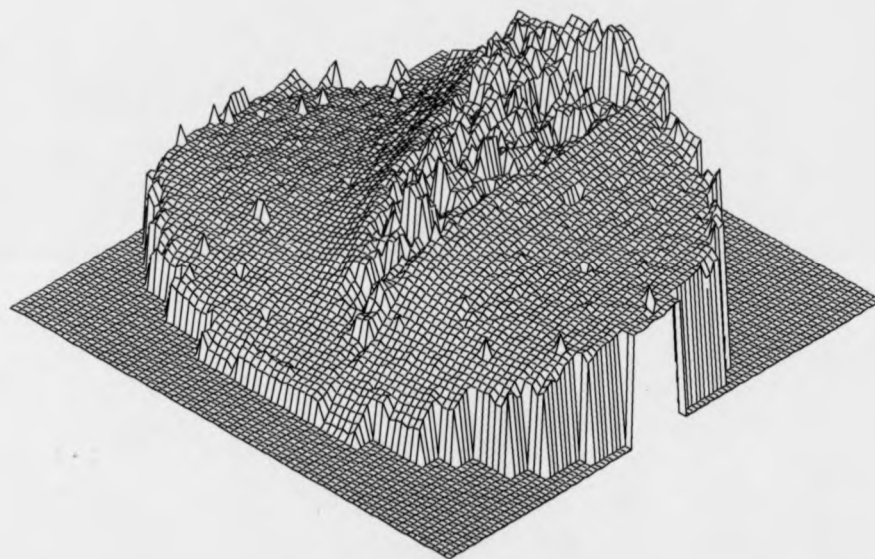


Figure 5.36: Mesh Plot Of Soldering Iron Solution

5.11 Conclusion

This chapter has given a number of examples of fringe analysis, and the application of the MSTT phase unwrapping strategy. This chapter concludes the application of digital image processing to fringe analysis.

The next chapter considers the application of digital image processing to Particle Image Displacement Velocimetry (PIDV).

Bibliography

- [1] T. R. Judge, C. Quan, P. J. Bryanston-Cross, "Holographic Deformation Measurements by Fourier Transform Technique with Automatic Phase Unwrapping", Optical Engineering, to be published.
- [2] P. Hariharan and B. S. Ramprasad, "Wavefront tilter for double-exposure holographic interferometry," J. Phys. E: Sci. Instrum. 6, pp. 173-175, 1973.
- [3] P. D. Plotkowski, Y. Y. Hung, J. D. Hovanesian, and Grant Gerhart, "Improved fringe carrier technique for unambiguous determination of holographically recorded displacements," Opt. Eng. 24(5), pp. 754-756, 1985.
- [4] D. R. Matthys, J. A. Gilbert, T. D. Dudderar, and K. W. Koenig, "A windowing technique for the automated analysis of holo-interferograms," Opt & Laser Eng. 8, pp. 123-136, 1988.
- [5] J. Takezaki and Y. Y. Hung, "Direct measurement of flexural strains in plates by shearography," J. Appl. Mech. 53, pp. 125-129, 1986.
- [6] C. Quan and P. J. Bryanston-Cross, "Double-source holographic contouring using fibre optics," Opt. & Laser Tech., 22(4), pp. 255-259, 1990.
- [7] J. D. C. Jones, M. Corke, A. D. Kersey, and D. A. Jackson, "Single-mode fibre-optic holography", J. Phys. E: Sci. Instrum., 17, pp. 271-273, (1984).
- [8] H. I. Bjelkhagen, "Fiber optics in holography", Proc. SPIE 615, pp. 13-17, 1986.

- [9] S. Timoshenko and S. Woinowsky-Krieger, *Theory of plates and shell*, McGraw-Hill, New York, 1959.
- [10] G. O. Reyndds, D. A. Servaes, L. Ramos-lzquierdo, J. B. BeVelis, D. C. Peirce, P. D. Hilton, and R. A. Mayville, "Holographic fringe linearization interferometry for defect detection," *Opt. Eng.* 24(5), pp. 757-768, 1985.
- [11] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Automatic interferogram analysis techniques applied to quasi-heterodyne holography and ESPI", *Opt. & Lasers Eng.* 14, pp. 239-281, 1991.

Chapter 6

Particle Image Displacement Velocimetry

6.1 Introduction

This chapter turns the discussion of quantitative image analysis away from interferometric applications. It discusses the application of image processing to Particle Image Displacement Velocimetry (PIDV), where small particles are imaged to produce velocity information from a flow field.

In the applications considered a double exposure is made of each particle so that the resultant image records particle pairs, where the displacement between the particle images encodes velocity. A spatial pairing analysis technique has been evolved which attempts to detect and measure velocity from individual particle pairs. This strategy has been adopted because of the sparse seeding found in high speed PIDV experiments. It is difficult to place the seeding particles into high speed flows in sufficient quantities to justify the use of more standard, but time consuming, statistical processing such as Auto-correlation. An advantage of the spatial pairing strategy is that the original image and the vector results may be overlaid and compared. This aids understanding of the flow structure.

The work described in this Chapter was partly undertaken during a LINK scheme. The partners in this scheme being, SERC, DTI, Rank Cintel, the Aircraft Research Association (ARA) and Warwick University. The LINK scheme was extremely successful. Its original aims were defined as follows,

- i) To provide a family of optical diagnostics which could be used at transonic flow speeds to make non-intrusive measurements within a 'hostile' industrial environment such as that presented by the ARA Transonic Wind Tunnel (TWT).
- ii) To consider the most relevant type of technology which could be transferred into a company with an aerodynamic testing background, such as that found in ARA, but with only a limited infrastructure to support optical diagnostic methods.
- iii) To generalise the optical diagnostic methods such that they could eventually become self standing measurement techniques.
- iv) To break ground scientifically in areas which are critical to the development of quantitative flow visualisation for the next generation of transonic wind tunnel testing.

PIDV was one of four diagnostic techniques developed. The others being Laser Light Sheet, Holography and Wing Deflection Measurement. For a summary of this work see reference [1]. The author's part in the scheme was to provide digital computer control during the tests and quantitative analysis of the images obtained. Digital control required software synchronisation of laser, cameras and frame store.

An excellent and comprehensive introduction to the subject of PIDV is given by Adrian in reference [2] in a lecture series given by the von Karman Institute in 1988.

Adrian describes Particle image displacement velocimetry (PIDV) as a method of measuring many fluid velocity vectors simultaneously, over extended regions of a flow domain. The intent is to combine the accuracy of single-point methods such as laser Doppler velocimetry with the multi-point nature of flow visualisation techniques. Efforts along these lines have been undertaken by numerous research groups during the past decade. PIDV is one of several approaches that have been aimed at measuring accurately velocities at hundreds of points in two-dimensional or three-dimensional regions.

When the flow field is unsteady, multi-point measurement techniques are capable of creating instantaneous pictures of the flow field that are unavailable from single-point measurements. Such information is much needed in the study of turbulent flow, where it is now widely recognised that the instantaneous realisations of the flow may bear little resemblance to the average structure. The need to study instantaneous unaveraged coherent structures has been one of the primary motivations for the development of multi-point measurement methods.

A characteristic of turbulent flows is that they contain a range of motions at a variety of scales. Experimental techniques must therefore be able to capture large coherent structures as well as having sufficient spatial resolution to capture small scale features. [2]

The marker used to seed the flow in PIDV is a small optical scattering site - such as a solid particle or gaseous bubble in liquid flow, or a solid particle or liquid droplet in gaseous flow. Gaseous air flows are considered here. The seeding employed has been either water droplets or solid latex particles.

There are a variety of ways of encoding velocity in the images. The approach used here was to employ a pulsed laser to produce a series of pulses. This in turn produces a series of images of each particle. In fact two pulses were normally used so that the resultant images contained 'particle pairs', that is two images of each particle. Some other possible coding schemes are shown in Figure 6.1, taken from reference [2].

Illumination for the project took the form of a light sheet, see Figure 6.2. This is a thin sheet of light illuminating a planar region, of the order of a few millimetres thick.

Under the LINK scheme PIDV was applied in a hostile industrial environment at a much larger distance than had previously been demonstrated. It proved to be a powerful technique. The technique has been seen to be within the capacities of a test centre to operate on a regular basis, without the constant attendance of expert practitioners. The velocities of 1 micron particles, travelling at transonic speeds, were measured at an optical distance of 2.4m.

A variety of improvements to the optical system were made in the course of the LINK scheme. The optical developments were made by the optics

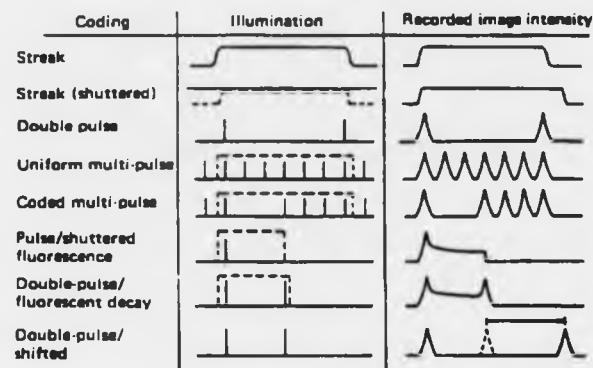


Figure 6.1: Coding Techniques for Image Velocimetry

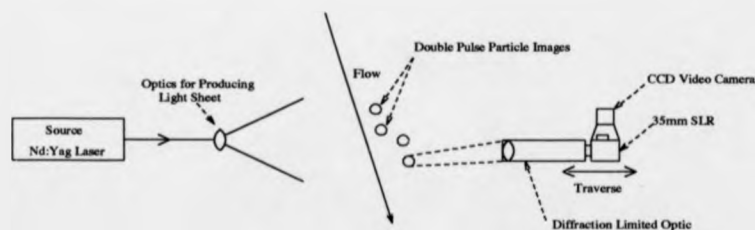


Figure 6.2: Schematic Diagram of PIDV as Applied in Transonic Wind Tunnel

team of P. J. Bryanston-cross, C. E. Towers and D. P. Towers. A single oscillator double pulse Nd/Yag laser was the final choice of laser. This was of lower power than the Ruby originally employed, but permitted a much higher repetition rate (double pulse to double pulse). The drop in power of the laser was found to be acceptable from initial tests with the Ruby. The Nd/Yag laser was capable of giving a maximum of a tenth of a Joule in energy per pulse, whereas the Ruby could give up to a Joule. The repetition rate of the Ruby was around 30 seconds, however, whereas the Nd/Yag could repeat at around 10Hz.

Double-pulsed lasers such as the Ruby and Nd/Yag produce high energies in pulses of very short duration, of the order of 10 to 25 ns. Because the particles being imaged in the TWT were so small the order of power supplied by such lasers was necessary. This is related to the light scattering properties of small particles (the Mie light scattering theory is described in detail in reference [3]). The particles have to be small to follow the flow correctly, and not simply continue on their own paths when shocks occur, for example. The subject of visualising such small particles in the application of PIDV has been explored by Bryanston-Cross [4]. The frequency doubled Nd/Yag laser employed could produce two 10ns 0.1J pulses separable over the range 50nsec to 100 microsecs. It had a wavelength of 532nm. Several properties of the laser helped make the PIDV tests successful. These are described by Dr. Bryanston-Cross in reference [5], and given below,

- i) The solid state Nd/Yag pulse has a well defined Gaussian beam profile, which could be focused to produced a sheet of laser light 0.3mm thick. The quality and width of the light sheet were confirmed by placing a piece of laser analysis paper in the plane of the sheet to produce a burn pattern.
- ii) The Mie theory shows that the size of a sub-micron particle is related logarithmically to its ability to scatter light. Thus halving the particle size from 500nm to 250nm theoretically produces an order of magnitude reduction in the scattered light which can be collected in the side-scatter mode. The frequency doubled Nd/Yag laser has a greater Mie scattering efficiency for sub-

wavelength particles than the alternative Ruby pulse laser which operates at 695nm.

- iii) The laser could be run in an open lase mode at a repetition rate of 10Hz. This meant that alignment of the optical system was a simple task. Sharp focus was essential because of the narrowness of the depth of focus of the optics and the light sheet.

[5]

The film used for this test series, TMAX 3200, has both high speed and high resolution characteristics. TMAX has a resolution of 100 lines/mm and can be used at a sensitivity of ASA (ISO) 1280. However, TMAX is insensitive to the near red and infra-red parts of the spectrum which means that it may not be used to record images made using a Ruby laser as the illumination source.

However, subsequent tests have been performed using Kodak high speed infra-red film. The speed and resolution of this product were found to be very similar to TMAX.

It is fairly clear that there is a limiting balance between the amount of light scattered from a particle and the speed and resolution of the film required to image it. It is also known that this is of particular importance when sub-micron particles are considered. The break through in reaching large operating distances came from a re-appraisal by Dr. Bryanston-Cross of some work done by Adrian. Adrian had derived a relationship, between the factors above, based on fundamental optical wave theory, but the derivation made use of an empirically derived factor which related to the particular lens used. Adrian did not take into account the effect of simple geometric lens aberrations as found in most conventional lenses. The result of this was that the potential sensitivity of the technique was underestimated [5].

From a detailed study of the optical performance of available camera lenses, conducted by Warwick jointly with the Royal Aerospace Establishment (RAE), it was determined that standard lenses imposed a significant resolution limit on the technique. By using diffraction limited imaging optics it was possible to extend the previous stand-off measurement distance of sub-micron particles from 100mm to 2.4m.

A 35mm camera was used to record the particle data on film, through the diffraction limited optics. A video camera, installed in place of the pentaprism, enabled remote focusing of the camera so that the narrow depth of field of the optic could be overlapped with the position of the light sheet, see Figure 6.2.

It was later seen that transonic Video PIDV results were obtained through this video camera. That is real time Transonic Video PIDV was observed in the control room as tests progressed. It was then realised that a video approach to Transonic PIDV was possible and desirable. It was also noted that the analysis techniques for Transonic Video PIDV would require a digital image processing method rather than the more common optical processing methods which have been applied to PIDV images on film, these methods are described later. Moreover real time image processing, which is becoming more accessible, could make vector velocity results available during the test and so allow interaction with the experiment.

One of the motivations for developing a digital processing method was in order that it might later be applied to Transonic Video PIDV. The field of view of the CCD camera employed was much less than that of the film camera, although the pixel size of the CCD detector and the grain size of the film were comparable. That is the film did not provide a greater spatial density of detector sites, only a larger number. Video cameras with large arrays of detector sites are available, but as yet these are expensive and have a frame transfer rate considerably lower than that of conventional video systems (of the order of seconds as against $\frac{1}{25}$ of a second). However, the potential for a high resolution video system was obvious. The wet processing stage in the film based technique takes too much time.

A more standard CCD may be used as a probe to seek active areas. These may then be made the focus of attention. The 35mm SLR and CCD may then be used in conjunction. The CCD seeking out the relevant flow features, and the film camera being used to record single frames with a wider field around such areas. These developments mean that Transonic Video PIDV offers real time results in the visualisation of complex transonic flows at very large stand-off distances.

6.2 History of Particle Image Displacement Velocimetry

Grant and Smith give a brief summary of the development of PIDV in reference [6]. The history below is loosely based upon this reference. PIDV was first described in papers by Grousson and Mallick [7], Barker and Fourney [8] and Dudderar and Simpkins [9]. Grousson and Mallick employed polystyrene spheres of $0.5\ \mu\text{m}$ diameter as a seeding material in their fluid, and an electrooptic modulator in the path of their 0.8 Watt, continuous wave laser to generate pulses of laser light. A cylindrical lens was used to generate the light sheet. The image of the fluid consisted of a speckle structure. Illumination of the fluid by two pulses left two mutually displaced speckle patterns upon the film. The displacement being different for different areas of the flow. The analysis technique employed was based upon a Fourier plane analysis. The application of PIDV in these first reports was to liquid flows. Experimental difficulties limited the speed of flows which could be investigated to speeds of the order of millimetres per second and over regions of the order of square centimetres.

Adrian and Yao [10, 11, 12] detailed the differences between the particle image and the speckle regimes and discussed the effects of particle scattering characteristics.

In reference [10] Adrian notes the difference between the techniques of recording multiple exposures of the speckle pattern translation during surface motion (for example, to measure in-plane displacements of surfaces, see ESPI series in Chapter 2), and applications involving fluids. He states that these are fundamentally different. The light scattering characteristics of fluids containing small particles can be quite unlike those of solid surfaces. For example, fluids are illuminated by a pulsed sheet of laser light whose thickness is Δz . Hence scattering occurs from a volume distribution of particle scattering sites rather than a surface distribution. The particles are typically small ($0.1 - 10\ \mu\text{m}$), and they act as discrete point sources of scattered light. The number density of particles per unit volume and their size can vary over a very wide range of values, depending upon the fluid and

its treatment. For speckle patterns to exist, the number of scattering sites per unit volume must be so high that many images overlap with random phase in the image plane. Since the number densities of scatterers in fluids can be quite low, it is possible that speckle is not present in many fluid applications, and that discrete images of particles are photographed instead. This, then, changes the mode of operation from laser speckle velocimetry to particle image velocimetry [10].

In concluding the same paper Adrian states that the source densities encountered in many air and water flows of interest in research and practical applications are often not high enough to produce speckle. He correctly notes that seeding in large scale flows or high speed flow becomes increasingly difficult and expensive as the concentration increases.

Meynart [13] first reported measurements in air where a pulsed Ruby laser was used to investigate an unexcited jet. Lourenco and Whiffen [14] described the use of a mechanically chopped, continuous wave, Argon-ion laser, and discussed the dynamic range of the instrument.

Bryanston-Cross first applied PIDV to transonic flows [4]. His work moved from a feasibility study at the Massachusetts Institute of Technology, to proving the technique in industrial environments, both in the course of the LINK scheme [1] and during tests at RAE Pyestock [5].

6.3 Processing Methods

The intuitively simplest processing method, for resolved particles, is direct analysis of the PIDV negative to determine the distance and direction through which the particles have translated between exposures. The problem then is to identify a particle and its partner. In densely seeded flows the probability of mis-matching particle images is high. One method that helps to resolve this problem is to pulse the laser several times, to create multiple images of each particle, which provides additional criteria for allocating particles to unique groups.

An alternative processing method demonstrated by Meynart uses whole field analysis of the image by optical Fourier transformation and filtering. This provides a pattern of fringes which represent iso-velocity contours.

The most appropriate method of analysis depends on the particle density within the image. In the application of high speed PIDV it is far more difficult to introduce the seeding material than in low speed applications. This means that the images produced from high speed PIDV are relatively sparse, when compared with those from low speed PIDV. Because of the sparse nature of the data, methods which rely upon local statistical averaging of many particle pairs are not appropriate.

It has been found that in order to believe in the measurements, in the presence of noise and laser light reflections (glare), it has been a requirement to be able to inspect the particle pair images themselves. This has been facilitated by the use of a video overlay in which the computed velocity vectors are overlaid upon the original PIDV image. The computation of individual measurements for each particle pair have been found to be more appropriate than area averages in order to understand flow structures which would otherwise be averaged away.

6.3.1 Two Dimensional Correlation and Two-Dimensional Spectrum Analysis of Young's Fringe Pattern

The notes from the von Karman institute [2] present a detailed theoretical analysis of two general and powerful analysis methods; these are, full two-dimensional correlation in the case of the image plane and full two-dimensional spectrum analysis of the Young's fringe pattern in the case of analysis in the Fourier transform plane [2].

These statistical methods can be used to analyse the PIDV negative when there are many particles present; they rely on the fact that the displacements of markers within a sufficiently small interrogation spot will be nearly equal, so that the probability of detecting the mean displacement is high compared to random pairings of marker images.

In a double-pulsed system, correlation techniques determine the displacement which gives the greatest correlation between the first image and the second image.

Using the Young's fringe method each pair of images within the interrogation region produces a set of interference fringes in the far field. The fringe

spacing is inversely proportional to the spacing of the pair images. The orientation of the fringes is perpendicular to the direction of travel. If there are several images within the interrogation spot, then the average fringe pattern will be dominated by the fringes associated with the average displacement.

These methods may be implemented optically employing a transparency of the marker positions. This transparency is translated to different positions in front of a light source (He-Ne Laser), to select different spots, see Figure 6.3.

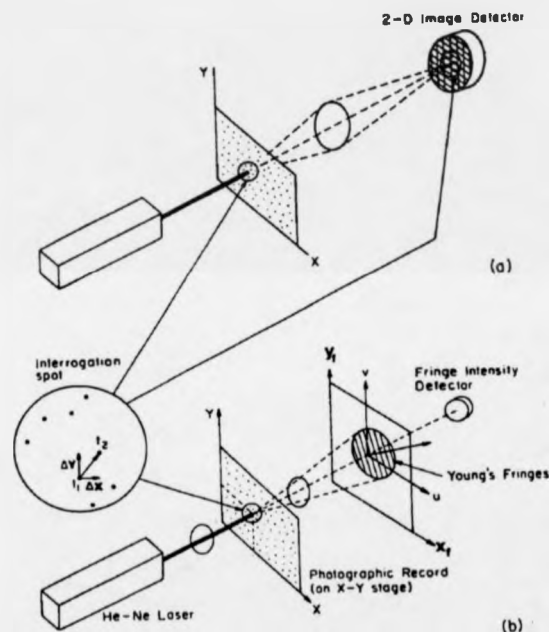


Figure 6.3: Interrogation of a Double Exposed Transparency (a) Processing in the 2-D Image Plane via Two Dimensional Correlation. (b) Processing by Young's Fringe Analysis in the Fourier Transform Plane of the Images

The correlation theorem is outlined below. Reference [15] gives a good introduction to Fourier transform spectral methods. Given two functions $h(x)$ and $g(x)$, the correlation of the two functions, denoted $\text{Corr}(g, h)$, in the continuous case is defined by

$$\text{Corr}(g, h) \equiv \int_{-\infty}^{\infty} g(x+a)h(x)dx \quad (6.1)$$

x is a spatial measure in this case. The correlation is a function of a , which is called the lag. The transform pair below permits conversion to and from the Fourier domain

$$\text{Corr}(g, h) \longleftrightarrow G(f)H^*(f) \quad (6.2)$$

given that g and h are real. This result shows that multiplying the Fourier transform of one function by the complex conjugate of the Fourier transform of the other gives the Fourier transform of their correlation. For the discrete case, of two sampled functions g_k and h_k , each with period N ,

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k}h_k \quad (6.3)$$

The discrete correlation theorem states that this discrete correlation of two real functions g and h is one member of the discrete Fourier transform pair

$$\text{Corr}(g, h)_j \longleftrightarrow G_k H_k^* \quad (6.4)$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j . The correlation of a function with itself is called its autocorrelation. In this case the discrete transform pair becomes

$$\text{Corr}(g, g)_j \longleftrightarrow |G_k|^2 \quad (6.5)$$

This is called the Wiener-Khinchin Theorem. Autocorrelation, via application of a digital two-dimensional Fourier transform and the above theorem, is employed to analyse an example later in this chapter. The power spectrum computed during this process resembles the Young's fringe pattern.

It is important in the application of correlation techniques to consider end effects, since the images will not be periodic as intended by the correlation theorem. Zero padding is employed. For example, if lags as large as $+$ or $-K$ pixels are sought, then a buffer zone of K zeros must be appended at the

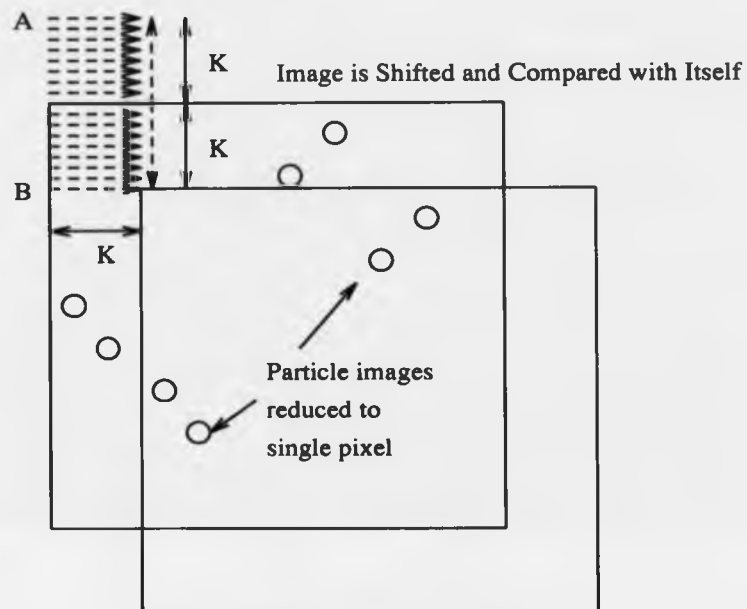
edge of the image. If this padding is not applied then erroneous results will be obtained when data close to one edge of the image is compared with data close to the opposite edge. If all possible lags are sought, then this means appending a buffer zone with K equal to the size of the image.

Analysis by the Fourier correlation theorem serves to illustrate the relationship between the optical Young's fringe approach, and the digital Fourier methods. Researchers have tended to take this approach, of mapping the optical analysis method to the digital domain by simulating the optical process. That is, by producing the analogue of the Young's fringe pattern (the power spectrum) by using a 2D FFT. However, this has tended to blind researchers to the possibility of implementing the correlation technique in other ways. It seems that the huge advantages of a spatial implementation have been overlooked.

Myrman [16], for example, discusses the use of an NMX-432 array processor, in the analysis of Dr. Adrian's PIDV data, in order to perform the autocorrelation in the Fourier domain. The images analysed are from a 4×5 inch negative and contain 12,500 interrogation spots, each spot containing 64,000 pixels. Processing time is given as around 3 hours. He briefly discusses the time complexity of the Fourier approach when implemented upon a digital computer.


Pre-processing of the digitised PIDV negative is performed by Myrman, prior to the analysis. This pre-processing reduces the pixel image of each particle to a single pixel, calculated as the the centroid. The motivation for this is in order that the Fourier impression of the PIDV image is not contaminated by the transforms of the individual particle images. This contamination occurs as each particle differs slightly in shape and size, the pre-processing eliminates the image of each particle reducing them all to a single pixel. As is seen later, without this pre-processing large particle images can dominate the fringe field. This problem must exist, without proper filtering, in optical methods.

The pre-processing applied by Myrman is sensible (it is the same pre-processing as applied by the spatial pairing technique). However, Myrman fails to take advantage of the simplicity of the image thus obtained. For example, Figure 6.4 illustrates a spatial approach to the autocorrelation problem.



Autocorrelation implemented up to lag of K , by spatially:

- Bit shifting image.
- Bitwise AND operation.
- A count of set bits in resulting image.

 = Lags not required.

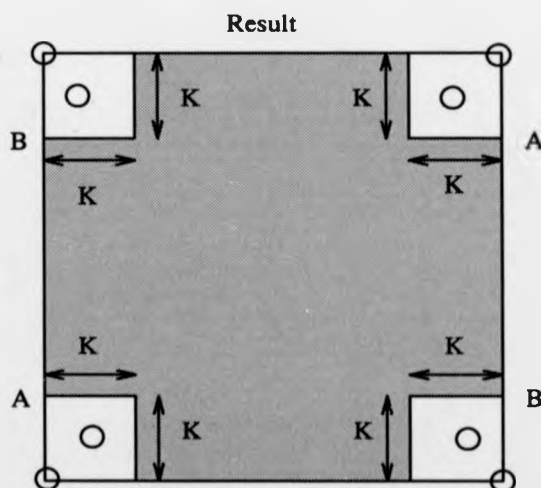


Figure 6.4: Efficient Use of Spatial Correlation

The spatial correlation technique may be implemented very efficiently over such an image. To begin with bit manipulation arithmetic may be used, as operations are then performed over a binary (bilevel) image. Perhaps more importantly, the complexity of the processing task may be vastly reduced by definition of a maximum value for the lag K . This means that the correlation process may be *halted* before all possible lags have been computed. This is an important point, the *global* transforms of the Fourier approach, which compute all possible lags, need not be performed. It is only necessary to compute a relatively small number of lags up to that corresponding to the maximum possible velocity in the image.

The result of the autocorrelation method is an average velocity vector for the area of the image considered. Individual pairs are not picked out as is the case with the spatial pairing method.

6.4 Initial Semi-Automatic Data Reduction System for Transonic PIDV via Spatial Pairing

The aim of processing is to produce a velocity vector field from the digitised particle field. The digitised particle field typically contains

- i) Images of the paired particles.
- ii) Single unmatched particles travelling perpendicularly to the light sheet.
- iii) Glare from laser light reflection.
- iv) Film grain noise.

The spacings of the paired particles must be identified and translated into velocities.

The particle data in the high speed flows considered is, as has been mentioned, sparsely distributed. One of the concerns was that a Fourier approach, for example, would be largely processing empty space and so have

become rather distant from the problem itself. A spatial pairing analysis strategy has therefore been evolved to deal efficiently with the sparse data fields encountered.

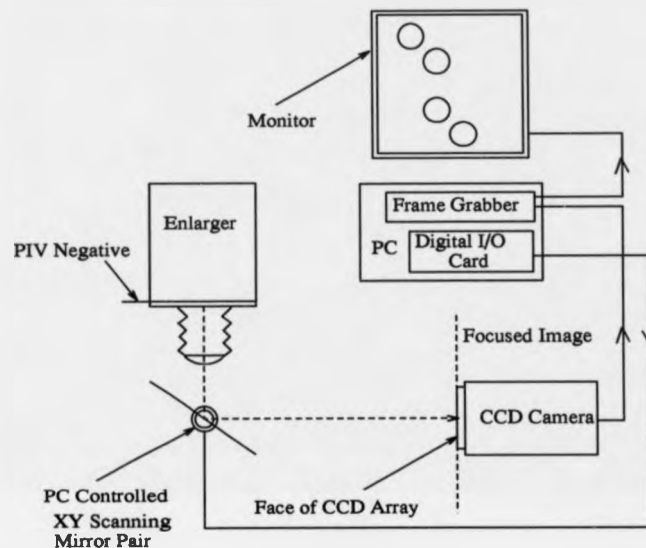


Figure 6.5: Initial Semi-Automatic Data Reduction System for Transonic PIDV

The first approach taken to the analysis of the PIDV data from the transonic tests is shown in Figure 6.5. This system involved projection of part of the PIDV negative on to the face of a CCD cell within a video camera. A typical image obtained from the system is shown in Figure 6.6. The image shows two clear particle pairs, and a possibly unpaired particle. The partner of the lone particle may be just visible within the film grain noise, or may have been lost as it passed beyond the light sheet. The image in this Figure is 512 by 512 pixels. There is an aspect ratio of 1.41 from the CCD camera to the frame capture board. This has been reproduced in the Figure. Each pixel in this image represents about 1 micron in the PIDV negative. The particle images are approximately 30 microns, or 30 pixels, across.

The question of accuracy is explored below. Suppose that an experiment has been conducted with a laser pulse separation of 2 microseconds and that the particles imaged were moving at 200 m/s. This would give a particle pair

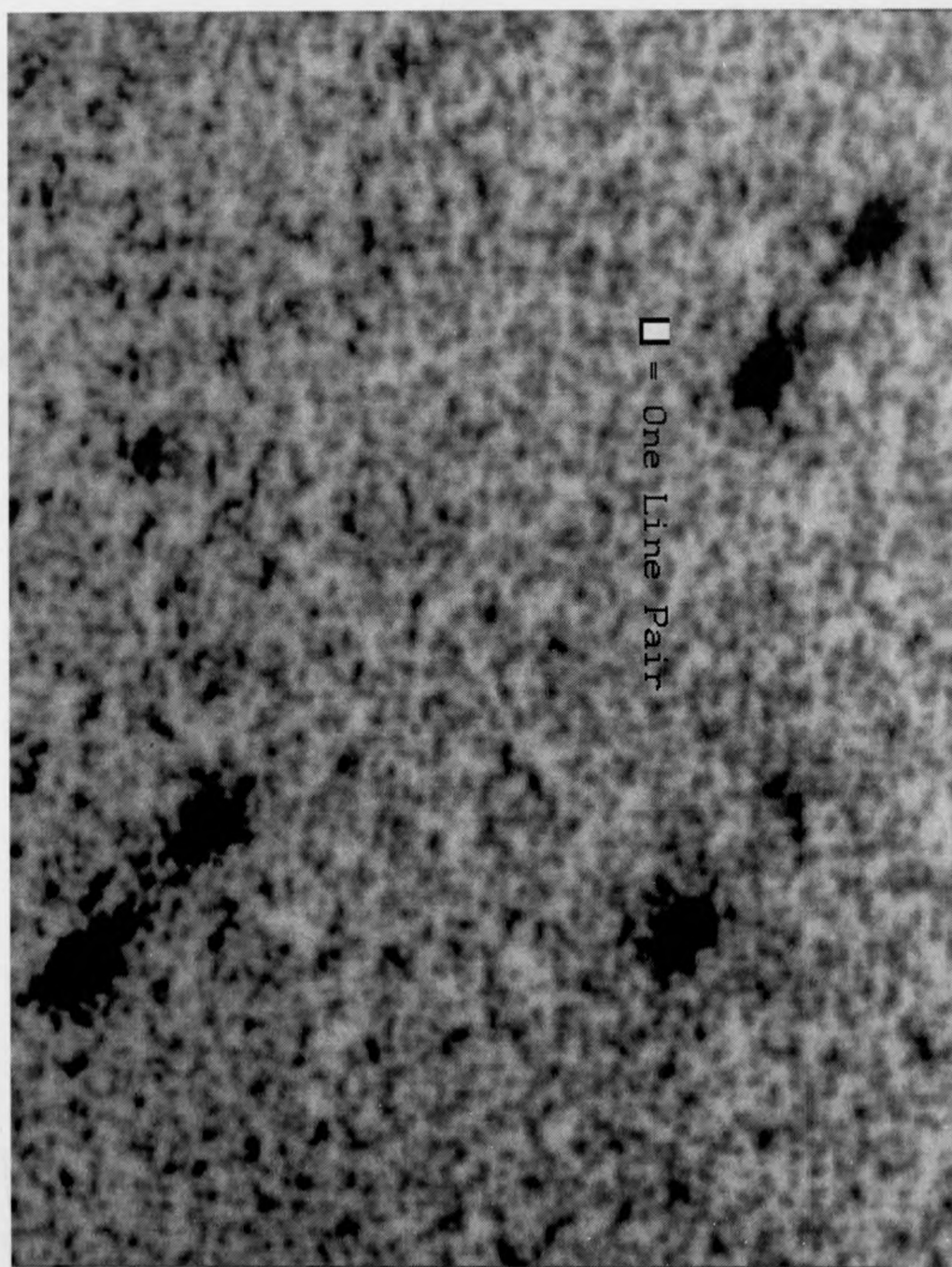


Figure 6.6: Image Captured from Original Data Reduction System showing Two Clear Particle Pairs

separation in space of 0.4mm. The actual image size on the negative would be 1.9 times smaller than this, a feature of the optical system. This produces an image displacement of approximately 210 microns. The individual particle image diameter as defined by the imaging lens employed, was found to be approximately 30 microns as illustrated in Figure 6.6 (in this Figure the particles are not moving at 200 m/s).

The resolution of the film is 100 line pairs/mm. That is 10 microns per line pair. The line pair resolution means that two adjacent lines can be separately resolved at a spacing of 10 microns on the film, it is therefore a somewhat conservative estimate for the resolution of the film. Figure 6.6 suggests that the line pair resolution does not directly represent the resolution with which the position of a particle may be determined.

It can be seen that the resolution of the digitisation process, in this system, is far above the resolution of the film, again see Figure 6.6. The limit of measurement accuracy is therefore in the film, and not in the digitisation process.

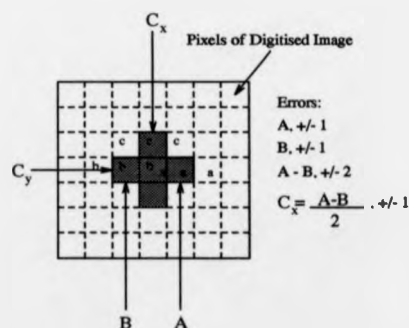


Figure 6.7: Computation of Particle Centre from Bounding Box in PIDV

Figure 6.7 illustrates the manner in which the centre of the particle is computed. A bounding box is used. The bounding box is found from the extreme x and y coordinates of the particle. Referring to Figure 6.7, A and B show the extreme x coordinates of the particle. An illustration of the calculation for the x coordinate of the particle centre, C_x is shown. C_y is calculated in a similar manner. The final error in the calculation of the position of the particle centre is a combination of x and y effects. Equation

6.6 describes this error e_c , in terms of the error in C_x , which is given as e_x and the error in C_y , given as e_y .

$$e_c = (e_x^2 + e_y^2)^{\frac{1}{2}} \quad (6.6)$$

Let us suppose that the position of the edge of a particle, in either the x or y directions can be determined to ± 0.5 line pairs (± 5 microns or approximately ± 5 pixels). e_x and e_y which represent the error in C_x and C_y , would then lie in the range from -5 to 5 microns, see Figure 6.7. The maximum value of e_c may be determined by substituting the extreme value 5 for e_x and e_y in Equation 6.6. The maximum error in e_c is seen to be $\sqrt{50}$ or about ± 7 microns. This worst error causes a measured displacement of the centre along a line at 45 degrees to the horizontal.

Suppose that the coordinates of the two particle centres in a pair are given by (x_1, y_1) and (x_2, y_2) , respectively. Equation 6.7 then describes the displacement d between the particles, in microns (or pixels). The equation also includes terms relating the error in measuring the positions of the particle centres. That is e_{x_1} is the micron (or pixel) error in measuring x_1 , e_{y_1} is the micron (or pixel) error in measuring y_1 , etc.

$$d = (((x_1 + e_{x_1}) - (x_2 + e_{x_2}))^2 + ((y_1 + e_{y_1}) - (y_2 + e_{y_2}))^2)^{\frac{1}{2}} \quad (6.7)$$

Using this equation it is possible to investigate the error distribution for the velocity measurement. Suppose that the x and y coordinates of the centres may be measured to ± 0.5 line pairs (± 5 microns or ± 5 pixels), consistent with the previous discussion on measuring the position of particle centres. The true spacing of the particles on the film is known to be 210 microns, for the example. However, the angle at which this velocity vector lies, in frame, has an effect on the accuracy of measurement.

The error distribution was first investigated by a brute force method. This method simulated the vector at angles from 0 to 90 degrees, and varied the errors e_{x_1} , etc (over the values $-5, 0, 5$ microns), in order to simulate the effect on d . This exploration yielded the plot of Figure 6.8. As can be seen the maximum error occurs at 45 degrees.

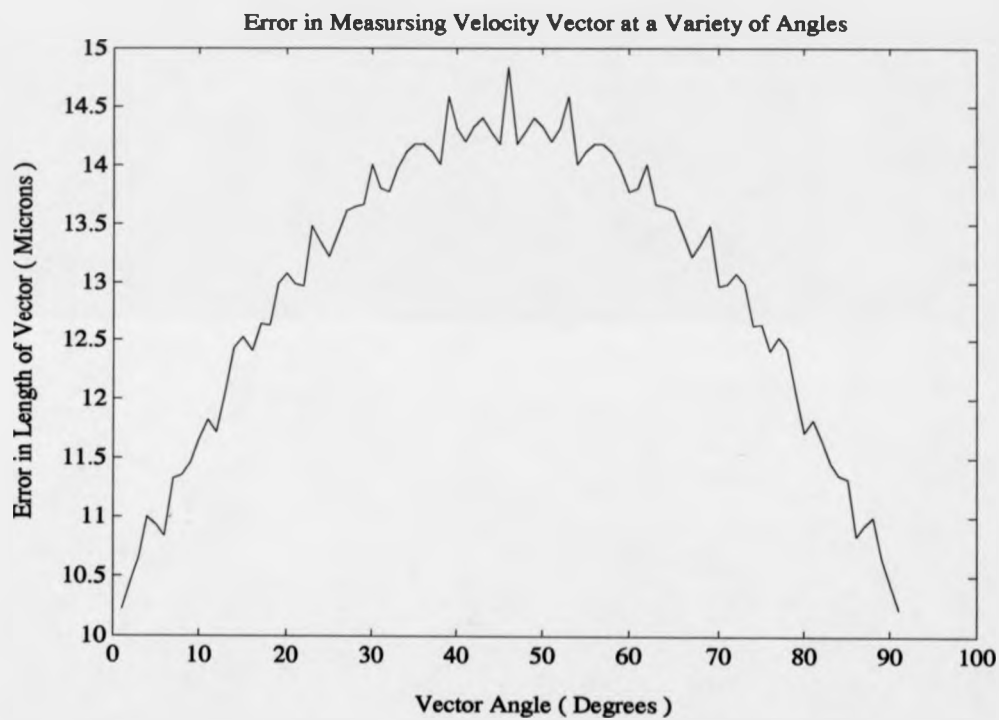


Figure 6.8: Error in Measuring Velocity Vector at a Variety of Angles



The maximum error in measuring the length of the velocity vector occurs when the vector is at 45 degrees.

At this angle the vector is aligned with the maximal particle centre displacement vector.

This means that the apparent length of the velocity vector is increased by a greater amount than at any other angle.

Compare a) and b)

Figure 6.9: Error in Vector Measurement is Greatest at 45 Degrees

This is explained by Figure 6.9. The displacement of the particle centres align with any vector at 45 degrees and so maximise the total length of the vector over all other angles. The error in length is almost entirely due to the displacement of the particle centres by 7 microns in opposite directions along the 45 degree line.

Provided the assumptions above are valid, this gives a total potential measurement accuracy, in the worst case, of about + or - 15 microns or 1.5 line pairs. This corresponds to a percentage accuracy for the example, at the film processing level, of + or - 7%, as 15 microns is 7% of 210 microns (the expected displacement for the example particle).

The part of the negative imaged by the system is controlled via a pair of computer controlled scanning mirrors, controlling x and y axis motion respectively. In this system, the operator was able to control movement across the negative via the cursor keys of the PC. The method of data collection involved the operator traversing the negative until particle pairs appeared on screen. The computer was then signalled to record the particle positions by processing the visible frame.

The positions of the particles in the frame were extracted via the following processing steps

- i) Thresholding of the image. This operation was applied to distinguish the particle images from the background. The operator was able to fix the threshold level to best extract the particle images from the background noise. This was not always entirely successful due to the similarity in intensity between the film grain noise and the particle images themselves.
- ii) Flood fill of marked areas. Those areas above the specified threshold level were flood filled in order to size them. That is the pixel area of each particle candidate was computed. It seemed that the area pixel count for patches of film grain noise was considerably less than the pixel count for particle images. During this processing phase a bounding box describing the spatial limits of each particle candidate was computed, as described earlier in this section, see Figure 6.7.
- iii) Those candidates which were more likely to be film grain noise than particles were rejected, based upon a user specified area threshold level.

The pairing process for this system was controlled by the operator. The computer aided the analysis by automatically recording the galvanometer coordinates of the scanning mirrors and the pixel coordinates of the particle centres. The centres were determined from the centres of the particle bounding boxes, as mentioned above. These factors were then converted to spatial coordinates and ultimately into velocity vectors using a specified pulse separation and scaling factor.

As can well be imagined the pairing operation was a time consuming and laborious process. A more automated analysis system was necessary. Rather than develop an improved system based around the hardware just described, a more compact and less labour intensive method of digitising the PIDV negative was adopted. This system overcame the problem of hysteresis. The galvanometers were under the control of 12 bit digital to analogue converters, the galvanometers did not always return to precisely the same point.

6.5 Improved Spatial Pairing PIDV Analysis System

This section describes the improved PIDV analysis system based around a flat bed scanner instead of the scanning mirror arrangement. In this system high resolution has been sacrificed for a more automated scanning process. It is noted that the resolution of the scanner over a 10 by 8 inch print is comparable to the line pair resolution of the film over a 35mm negative. That is 10 inches at a resolution of 300 pixels per inch gives 3000 pixels, whereas 35mm at a resolution of 100 line pairs/mm gives 3500 line pairs.

The image processing functions have been improved to provide a much more automated system. The processing strategy is described below.

The operator supplies band limits for the particle size. The program may either apply a series of iterations of the analysis procedure, varying tight band limits for velocity and direction, in order to build up a histogram of the velocity distribution, and from the peak of this histogram the chief velocity vector in the image. Alternatively they may specify their own band limits for velocity and direction and view the pairs which are found within those limits using a video overlay.

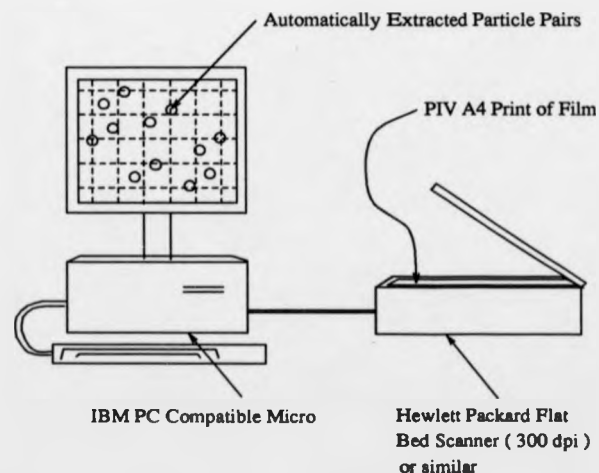


Figure 6.10: Data Reduction System Based Upon Flat Bed Scanner

The accuracy of the system depends upon a number of factors, the accuracy of timing of the delay between laser pulses, the relative component of velocity of the particle pair through the sheet and the digitisation process.

In the scanner system as shown in Figure 6.10, there is an intermediate photographic print process between capture of the PIDV data on film, and scanning of that print into the computer. Each step in film processing affects the accuracy of the system, as well as the scanning process.

It has been the practice to print the PIDV negatives as 10 inch by 8 inch prints. These are then scanned at 300 pixels per inch. In this way the whole field is digitised in one process. The resolution of the digitisation process could be increased by photographic enlargement of the PIDV negative, to the limit of the optical systems and film grain. The analysis system is not dependent upon the specifics of apparent particle size, in the digitised image, or scale, as these may be supplied as inputs.

Let us examine the accuracy of this system, using the same example, of a particle moving at 200 m/s with a pulse separation of 2 microseconds, as was used in the previous section. Suppose the 35mm negative is printed as a 10 by 8 inch print. The 35mm dimension of the negative maps to approximately 10 inches in the print (a more exact scaling factor is determined from a calibration shot). The spacing of the particle images in the print would be, $\frac{210 \times 10 \times 2.54}{35000} = 0.15$ cm, which is 6 hundredths of an inch. This corresponds to a displacement of 18 pixels in the digitised image. A particle with an image diameter of 30 microns on the negative would have a pixel diameter in the scanned print of about 3 pixels. The spatial position of the centre of the particle can be computed to + or - $\sqrt{2} = 1.4$ pixels by again applying Equation 6.6. If analysis of the scanned image produces a result to + or - 1.4 pixels, for each particle, then it produces a result to + or - 2.8 pixels for the measurement, as the position of each particle in the pair must be assessed. The analysis has a resolution of approximately plus or minus 1 part in 6.4, for the given example, which is 15.6%.

The accuracy per point is low for the example. However, the 15.6% figure was arrived at by considering a scan of the entire 35mm negative. Clearly, for this example, the accuracy achieved is unacceptable. A more accurate analysis could be achieved with the same system by scanning an enlarged

section of the 35mm negative. As mentioned above, there is a trade off between convenience and accuracy.

6.5.1 The Problem of Ambiguous Pairing and Comparison of Particle Positions

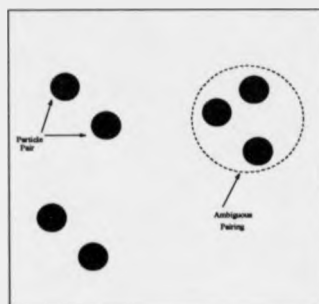


Figure 6.11: Schematic of Particle Pairs Showing an Ambiguous Pairing Problem

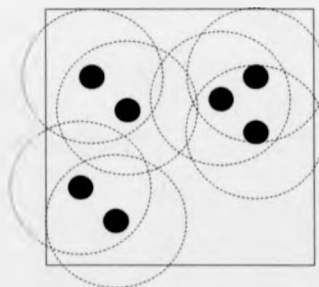


Figure 6.12: Schematic of Particle Pairs, Circles Define the Maximum Distance that a Particle could have Moved, Pulse to Pulse

One problem in the automatic analysis of PIDV data is the resolution of directional ambiguity. A coding scheme for the first and second particle is necessary. If this is not available then the analysis becomes confused in the area of a vortex or reversed flow region. In these cases there is a 180 degree ambiguity in the direction of fluid flow. Coding the image, perhaps by the

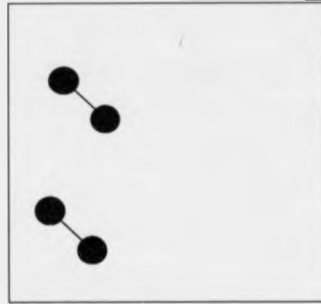


Figure 6.13: Schematic of Particle Pairs, Ambiguous Pairings are Deleted and Velocities Computed for the Remaining Pairs

use of two colour lasers can solve the problem, but such equipment was not available for the transonic tests performed to date.

A possible aid in determining the flow direction has been observed in the data gathered by the Warwick team. This relies on the fact that the power in separate laser pulses is seldom identical. That is, due to the difference in pulse energy between the first and second pulses, the image of the particle appears to change size in a predictable way between the first and second pulse. It may then be possible to assess the direction of motion by sorting the particle pair data using the size of the particle images. This effect can be seen in Figure 6.6.

Another problem for automatic analysis is that of ambiguous pairing. The problem of ambiguous particle pairings is illustrated by the series of Figures 6.11, 6.12 and 6.13. Figure 6.11 shows three particle pairs and a rogue particle whose pair is absent. This has made it difficult to tell which particle should be paired with which.

In order to first recognise this sort of situation a maximum value for the velocity of the particles is required. This, together with the pulse separation, allows the maximum distance that any particle could have moved between pulses to be calculated. This in turn allows the definition of circles of influence for each particle, within which partners must lie. This is illustrated by Figure 6.12. As can be seen, for the ambiguous pairing situation, on the right of the figure, each of the three particles lies within the sphere of influence of

two others. This is therefore an ambiguity.

Ambiguities are dealt with by deletion. That is all particles involved in ambiguous pairings are eliminated from consideration in the flow field solution. This is illustrated by Figure 6.13.

Velocity is constrained to lie within a specified band. The size of particles (that is their pixel area) is similarly constrained, and the angle of the velocity vector is also constrained.

The algorithm is able to reject glare in the image, or other artefacts which are too large to be particles, based on the specified limits for particle size. The sizing operation is performed via a flood fill of each distinct illuminated area.

The positions of those 'particles' which lie within the acceptance band for size are noted. A list is formed. Each particle is considered to have its pair within a distance band dictated by the minimum and maximum velocity. This band lies within two concentric circles centred on the particle.

A sort of the particle list is performed in order to group particles which could be pairs together. The original list is therefore subdivided into a series of sublists. Each sublist contains a grouping of particles.

At this point group lists which contain only one particle are removed. Group lists containing uniquely two particles are considered particle pairs. These form the basis for the velocity field solution. Groupings of larger numbers of particles are ignored.

Each particle is associated with a sub group. In order to perform this grouping operation a large number of comparisons between particle positions and particle groups must be made. This is time consuming. A strategy has therefore been developed in order to reduce the number of comparisons that need to be made. This involves computation of the spatially bounding box for each particle grouping. Instead of comparing each candidate particle to all of the members of any group, an initial test is made against the group bounding box, A in Figure 6.14. If the candidate particle is at such a distance from the bounding box that it could not be paired with any particle within the box, that is outside B in Figure 6.14, then the particle is not further compared with members of the group. Otherwise a more thorough comparison of the position of the new particle with those of the group is undertaken. This

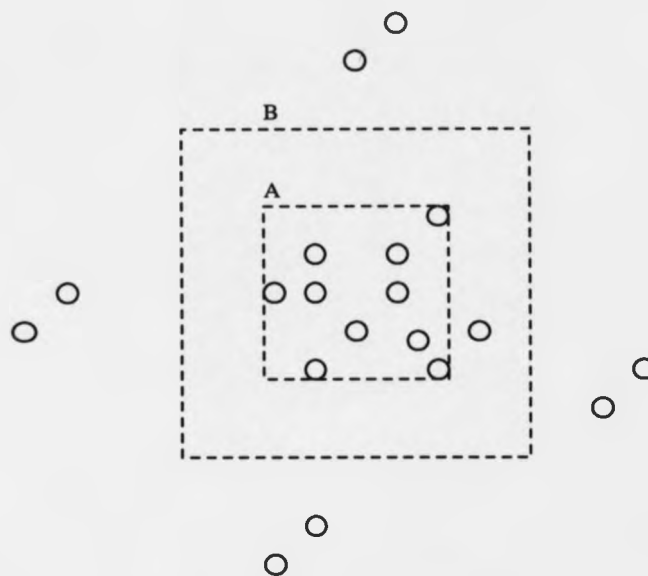


Figure 6.14: Bounding Box Test to Improve Performance of Pairing Algorithm

process considerably improves the performance of the algorithm.

6.6 Example of PIDV Analysis by 2D Autocorrelation Technique on a Single Pair of Particles

The simplest PIDV image open to analysis contains two unique particles. This case is given as a first example of analysis by the Autocorrelation method.

Figure 6.15 shows a pair of pixels, which represent a pair of particles. It will be seen later how reduction of particle images to single pixels aids the analysis procedure by eliminating the transforms of the individual particles, as mentioned earlier in reference to Myrman [16]. Figure 6.16 shows the 2D Power spectrum of Figure 6.15. The Figure resembles a Young's Fringe pattern. This image is squared and the inverse 2D FFT computed to give the

autocorrelation, according to the Wiener-Khinchin Theorem. This inverse application of the 2D FFT produces Figure 6.17. Here it is seen that the pixel particle images have produced two clear peaks, at a spacing of double the original particle spacing. The spatial positions of these peaks may be extracted automatically by a search for the two most prominent maxima to find the spacing of the pixels in the original Figure 6.15, clearly a trivial example. However, the same (or at least a similar) process may be applied in a real image, with many particles as will be seen below.



Figure 6.15: Single Pixel Pair Image

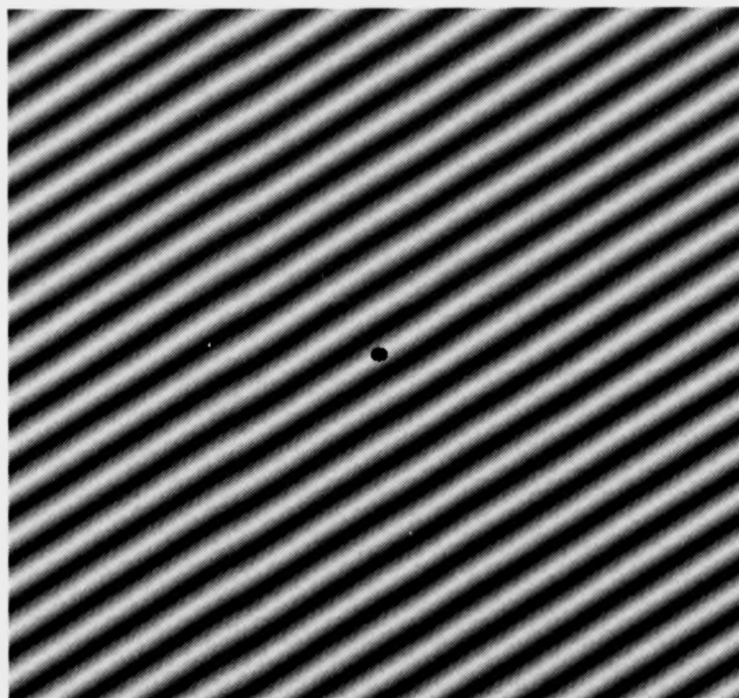


Figure 6.16: Single Pixel Pair Image 2D Power Spectrum with DC Removed

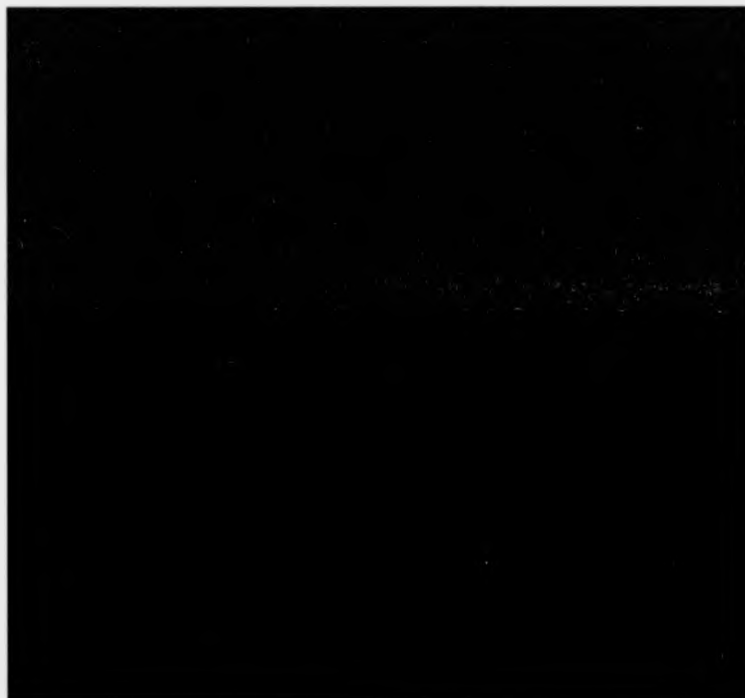


Figure 6.17: Single Pixel Pair Autocorrelation

6.7 The 2D Autocorrelation Method Applied With and Without Preprocessing of the PIDV Image

A comparison is made between correlation analysis without pre-processing of the PIDV image and with a pre-processing step. The pre-processing step reduces the particle images to single pixels defined by the centres of the particles' bounding boxes.

The data is taken from an experiment to measure the performance of the thrust reverser of a jet engine, see Figure 6.18. The final results of this test are to be published in ASME. The model employed was $\frac{1}{10}$ scale.

Micron sized water droplets were used as the seeding material. PIDV was used to make a specific measurement in the vicinity of the thrust reverser's 'kicker' plate to map the velocity and direction of the exit flow.

Figure 6.19 shows a scanned and reprinted version of the original PIDV photograph, with a box added to denote the area of the test image. Figure 6.20 shows the test image to be analysed.

The Autocorrelation technique is applied here using 2D Fourier Transforms. Following application of the technique, the correlation peaks may be found by searching about the centre of the field within two concentric circles whose radii are defined by the minimum and maximum lag, which correspond to the minimum and maximum velocities expected in the image.

Let us consider the 2D Power spectrum of the test image, shown in Figure 6.21. The test image itself is shown in Figure 6.20. At first sight this image appears to present a fairly clear fringe pattern. However, after the inverse transform, which gives the autocorrelation, Figure 6.22, it can be seen that the relatively clear low frequency fringes in Figure 6.21 have, in fact, been generated by the image of two abnormally large water droplets. These can be seen in the top left hand corner of Figure 6.20. The maximum band limit for velocity, 300 metres per second (equaling a lag of 31 pixels), is marked on the autocorrelation image as a white circle (not part of the transform). The large droplets can be seen echoed in the autocorrelation image in several places, most noticeably to the left and right of centre, beyond the

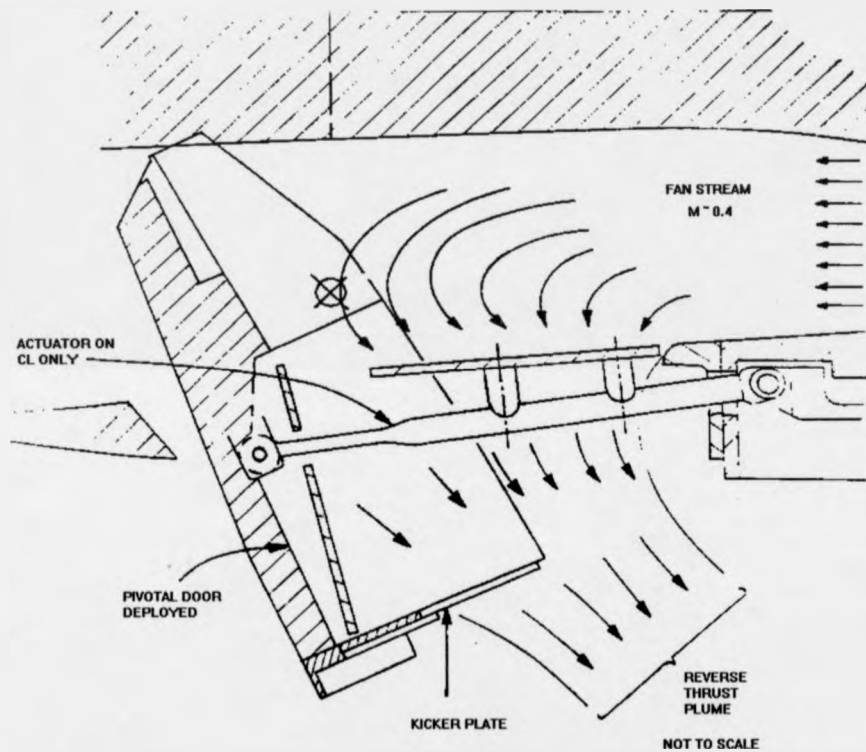


Figure 6.18: Thrust Reverser



Figure 6.19: Whole Field View of PIDV Experiment on Thrust Reverser
showing Area of Test Image in Box

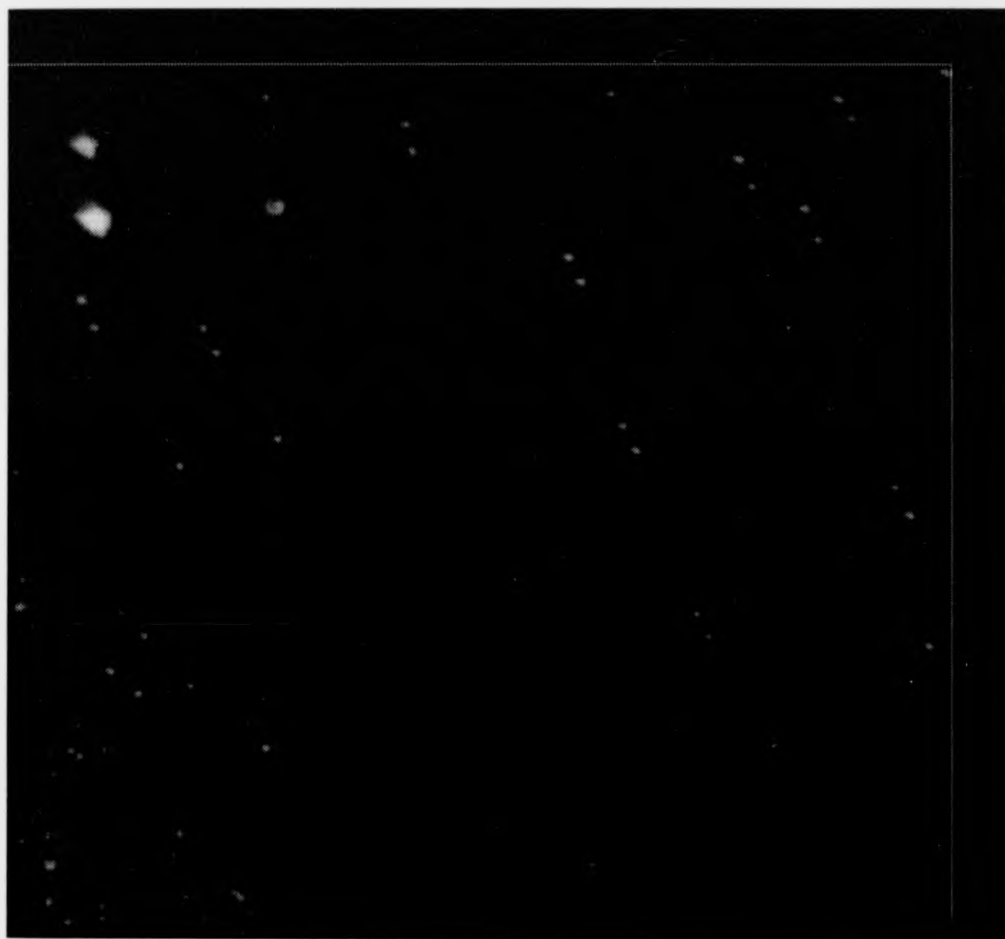


Figure 6.20: Test Image



Figure 6.21: Test Image 2D Power Spectrum with DC Removed

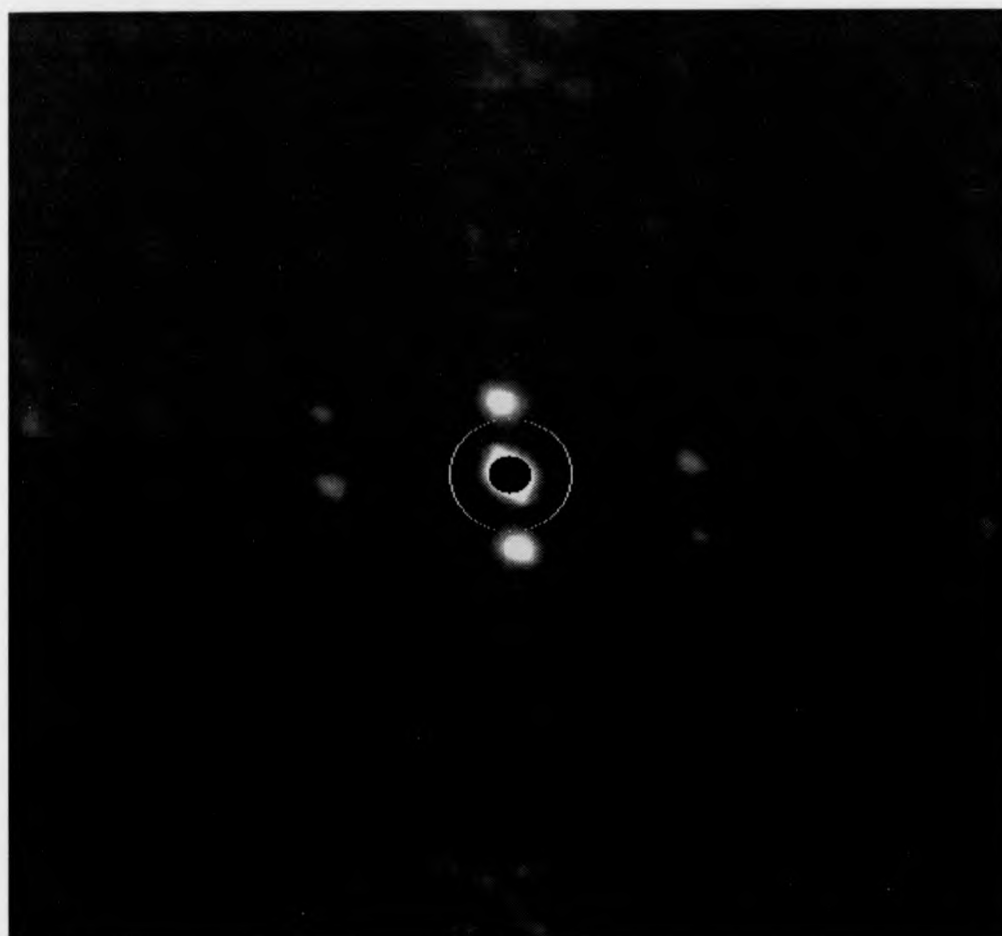


Figure 6.22: Test Image 2D Autocorrelation (Circle Defines Maximum Velocity 300 m/s)

area defining the maximum lag. Here the droplets have correlated with their own images.

The peak due to the velocity of the particles is spread. It lies near the centre of the Figure 6.22 and is difficult to detect. The peak has been spread by the transforms of the particle images. The analysis has been disturbed by the images of the seeding particles.

This is a serious problem and must be addressed before the analysis technique may be employed with any confidence. It would be a difficult problem to deal with if the transform were being applied in an optical system. Fortunately in a digital environment, pre-processing of the image is possible to eliminate the images of the particles in the transform.

It becomes necessary to apply a pre-processing stage to the image, in order to locate and size the individual particles. In the course of this processing 'particles' which are too large to follow the flow may be removed along with other artifacts. This is, in fact, the same pre-processing as is required for the spatial pairing analysis method.

Once this pre-processing has been applied an image of the type shown in Figure 6.23 is obtained. As can be seen the particle images have been reduced to the same normalised form of a single illuminated pixel. Abnormally large particles have been removed. The grey line at the top and right sides of the image defines the area of zero padding. The 2D power spectrum of this image is shown in Figure 6.24. This image shows a very noisy Young's fringe pattern. However, there is a discernible pattern of fringes in the direction of particle motion. The lack of clarity in this image will be explained by the following discussion.

First of all consider the inverse transform of Figure 6.24 to Figure 6.25. In this image, the autocorrelation of Figure 6.23, the minimum and maximum velocity band has been represented by the addition of two concentric circles, the inner circle defining the lower velocity band limit of 100 metres per second and the outer circle defining the upper limit of 300 metres per second. It can be seen that within these limits there is an expected and relatively clear pair of peaks for the perceived direction of particle motion. These peaks can be seen better in the contour and mesh plots of the central region of Figure 6.25 in Figures 6.26 and 6.27.

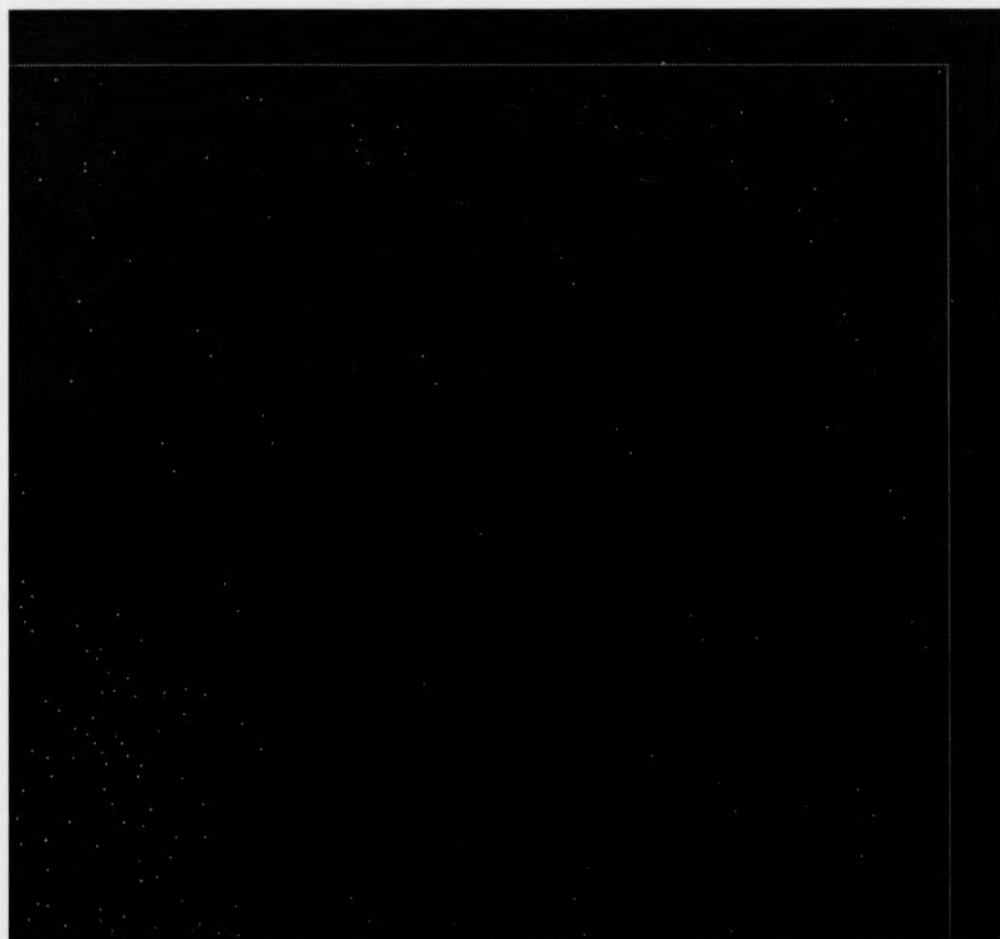


Figure 6.23: Test Image after Reducing Particle Images to Single Pixels



Figure 6.24: Reduced Test Image 2D Power Spectrum



Figure 6.25: Reduced Test Image 2D Autocorrelation (Circles Define Minimum Velocity = 100 m/s and Maximum Velocity = 300 m/s)

However, there is also another pair of peaks, which might not have been expected, running at about 45 degrees to the horizontal axis. The explanation for these twin peaks is that there are in fact two prominent flow directions and velocity profiles in the test image of Figure 6.20. This is not at all apparent from a first inspection of the image. The upper and right halves of the test image are responsible for the pair of peaks which are nearer vertical (in Figure 6.25). The bottom left hand corner of the test image is responsible for the other pair. This is a good example of the power of the Autocorrelation technique to pick out periodicities which are not apparent to a human observer and which are difficult to detect by other means.

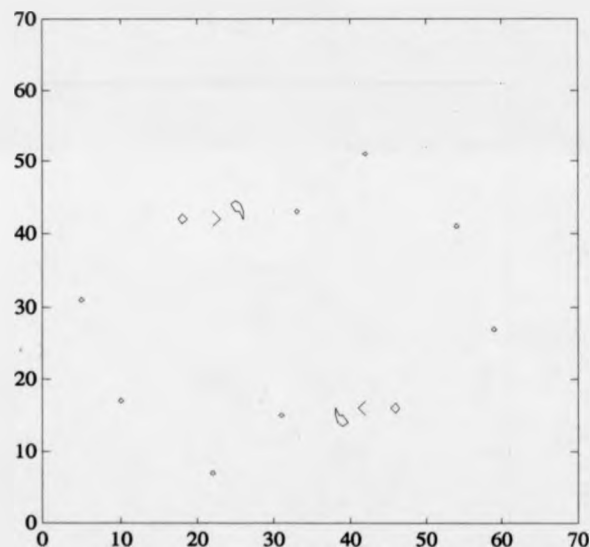


Figure 6.26: Contour Plot of Central Region of Autocorrelation for Reduced Test Image

Some evidence for this explanation is provided by the next series of images. The test image has been edited. The bottom left hand corner of the image has been cleared, and the analysis re-applied. The partial image is shown in Figure 6.28. The Young's fringe pattern of Figure 6.29, is much

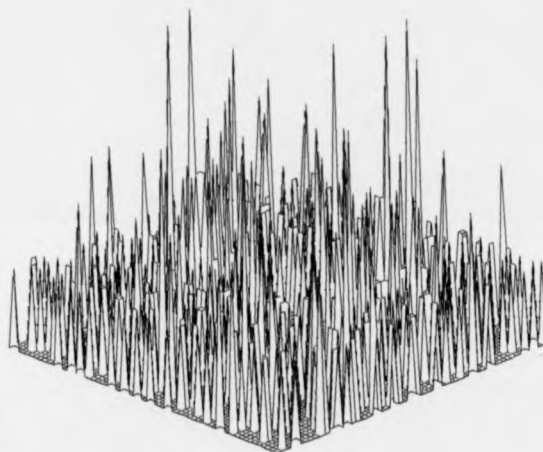


Figure 6.27: Mesh Plot of Central Region of Autocorrelation for Reduced Test Image

less noisy than previously, compare Figure 6.24. As can be seen from Figure 6.30, there is now only one clear pair of peaks. A contour map and mesh plot of the central portion of Figure 6.30 are shown in Figures 6.31 and 6.32 respectively.

6.8 Comparison of Analysis by 2D Autocorrelation with that by Spatial Pairing

The main velocity vector for the partial image is given in Table 6.1. The result for the other correlation peak in Figure 6.25, corresponding to the lower left corner of the image, is shown in Table 6.2.

Velocity = 157 m/s
Vector Angle = 153 deg
Displacement Between Peaks = 32.6 pixels

Table 6.1: Summary of Fourier Analysis Result for Edited Test Image

Velocity = 183 m/s
Vector Angle = 133 deg
Displacement Between Peaks = 38.2 pixels

Table 6.2: Summary of Fourier Analysis Result for Lower Left Corner of Test Image

Mean Velocity = 156 m/s , s.d. = 19.4 m/s
Mean Vector Angle = 152 deg, s.d. = 6.3 deg
Mean Particle Size = 16 pixels, s.d. = 12.9 pixels
Mean Particle Displacement = 16.2 pixels, s.d. = 2.0 pixels

Table 6.3: Summary of Spatial Pairing Analysis Result for Test Image

The spatial pairing result is shown in Figure 6.33, and Table 6.3. The result shown is as displayed by the AP analysis package, on a PC compatible computer (see Appendix on AP package). It is a video overlay, where the



Figure 6.28: Part of Test Image after Reducing Particle Images to Single Pixels



Figure 6.29: Part of Reduced Test Image 2D Power Spectrum with DC Removed



Figure 6.30: Part of Reduced Test Image 2D Autocorrelation (Circles Define Minimum Velocity = 100 m/s and Maximum Velocity = 300 m/s)

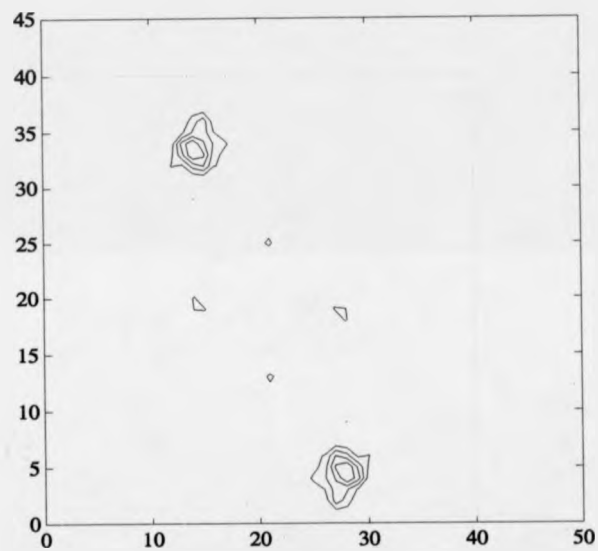


Figure 6.31: Contour Plot of Central Region of Autocorrelation for Part of Reduced Test Image

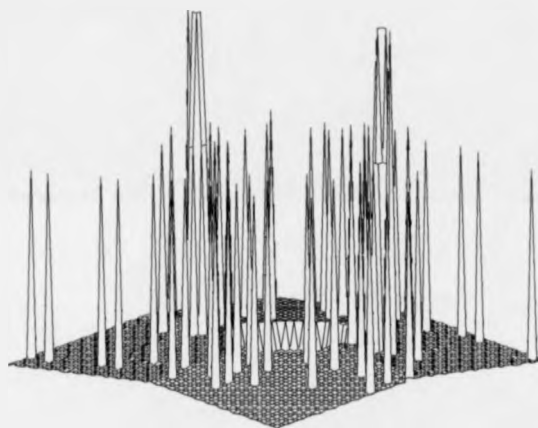


Figure 6.32: Mesh Plot of Central Region of Autocorrelation for Part of Reduced Test Image

velocity vectors are superposed on the original PIDV image. The author believes that this gives a better idea of what is happening in the flow than the pair of average velocity vectors given by the Autocorrelation method.

A 512 by 512 pixel section of the original field has been selected for this comparison test. A larger image would have proved unwieldy for processing by the FFT. However, the spatial pairing technique is easily applied to larger images, as is shown by the analysis of the full image given later in this chapter. The whole field of Figure 6.19 is processed in around 3 minutes by the technique on a 25MHz PC with a floating point coprocessor.

6.9 Discussion of Results for Spatial Pairing and Correlation Techniques

The two flow directions isolated by autocorrelation can be seen in the spatial pairing analysis result, although they are not highlighted so well. Examine and compare the direction and velocity of the vectors in the lower left corner of Figure 6.33, with those in the rest of the image.

The results for the spatial pairing analysis technique and the correlation technique are similar. The choice of technique should depend, in part, upon computational efficiency. As was discussed earlier the Autocorrelation technique can be considerably optimised using a spatial approach.

The number of comparisons required to analyse the field is the key to efficiency. The spatial pairing technique requires fewer comparisons than the Fourier correlation technique.

The spatial pairing technique operates over a sparse matrix. That is, the data operated upon is a list of coordinates rather than a two dimensional array of pixel values. Less data is being manipulated, and this may lead to a corresponding saving. In comparison to the Fourier correlation approach, every particle position is not compared with every other, particles are often compared with groups as discussed in Section 6.5.1. It would be interesting to investigate the performance of the spatial correlation technique against the spatial pairing technique. This is a subject for further work.

The spatial pairing analysis result for the whole image is presented in

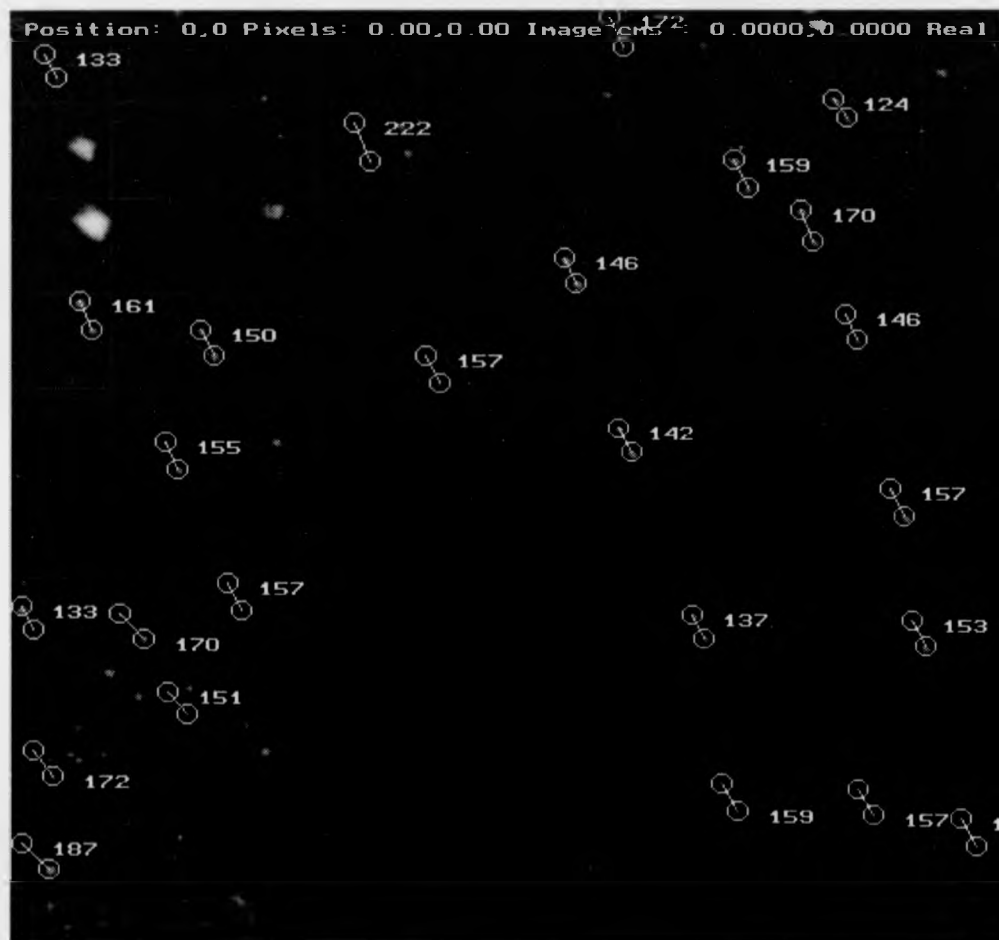


Figure 6.33: Result of Test Image Analysis by Spatial Analysis Superimposed on Test Image

the next series of Figures, that is not just the section which the test image represents. Figure 6.34 shows a plot of the velocity vectors. Figure 6.35 shows a histogram of velocity. Figure 6.36 shows a histogram of velocity vector angle. Table 6.4 shows a table summarising the result for the complete field.

Mean Velocity = 165 m/s , s.d. = 40.8 m/s
Mean Vector Angle = 150 deg, s.d. = 10.8 deg
Mean Particle Size = 16 pixels, s.d. = 15.0 pixels
Mean Particle Displacement = 17.2 pixels, s.d. = 4.2 pixels

Table 6.4: Summary of Spatial Pairing Analysis Result for Thrust Reverser (Complete Field)

6.10 An Example of High Speed PIDV at a Large Stand Off Distance (Applied in the Transonic Wind Tunnel of ARA Bedford)

This section gives a further example of the spatial pairing analysis method, used to analyse PIDV data obtained during the Link scheme.

Figure 6.37 shows the original PIDV photograph scanned at 300 pixels per inch. Figure 6.38 shows the velocity vectors resulting from the spatial pairing analysis. Note that the results near to the glare region are questionable. However, this result was obtained automatically, once some band limits had been specified. Velocity in this case has been constrained between 50 and 100 metres per second, and the vector angle between 60 and 120 degrees. The histograms show these to have been reasonable limits, Figure 6.39 shows a histogram of velocity and Figure 6.40 shows a histogram of the velocity vector angle. Table 6.5 shows a table summarising the result.

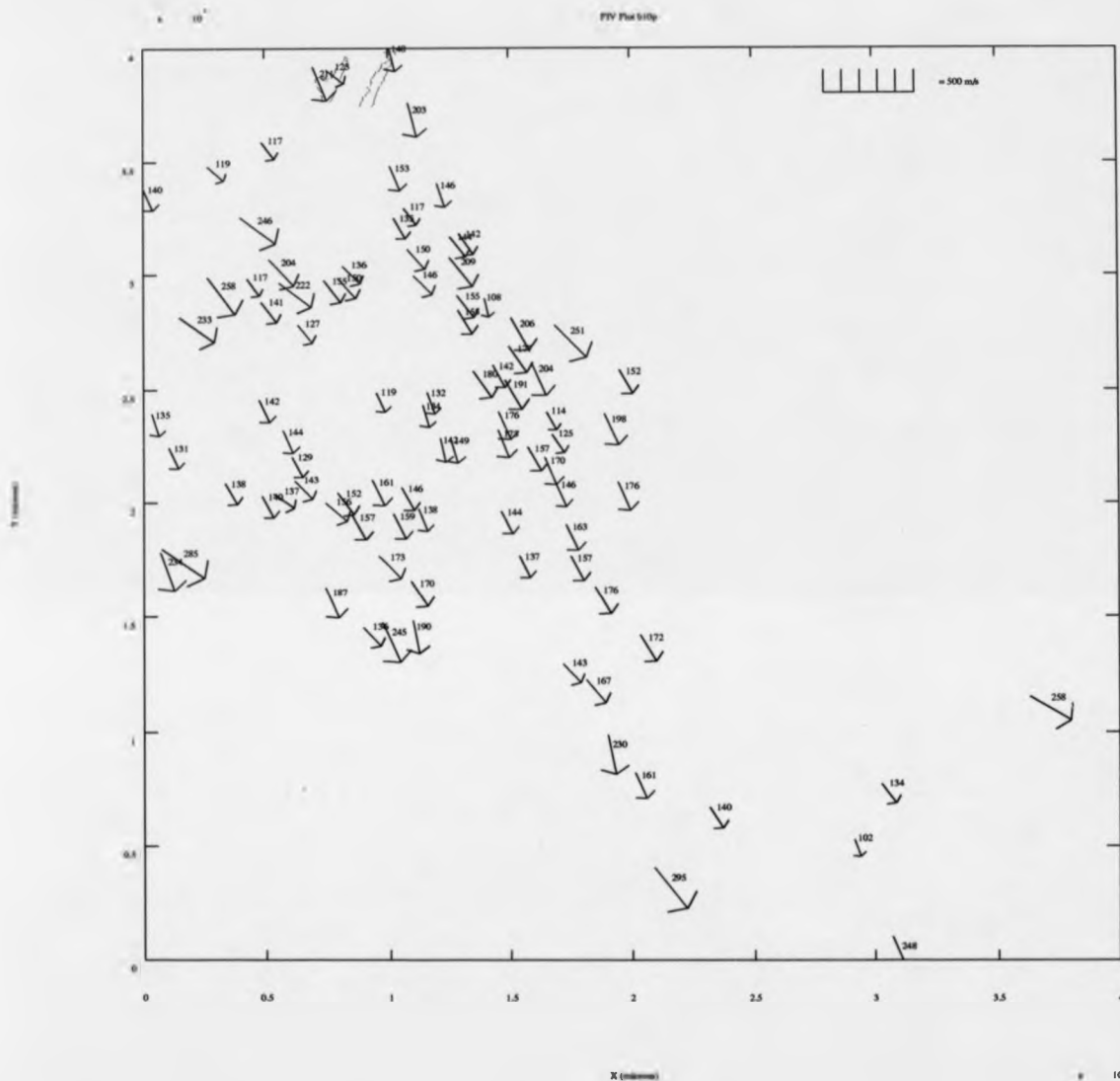


Figure 6.34: PIDV Vector Plot For Thrust Reverser, Pulse Separation 2.0 Microseconds, Minimum Allowed Velocity = 100 m/s, Maximum Allowed Velocity = 300 m/s, Minimum Allowed Angle (from vertical) = 120 degrees, Maximum Allowed Angle = 170 degrees

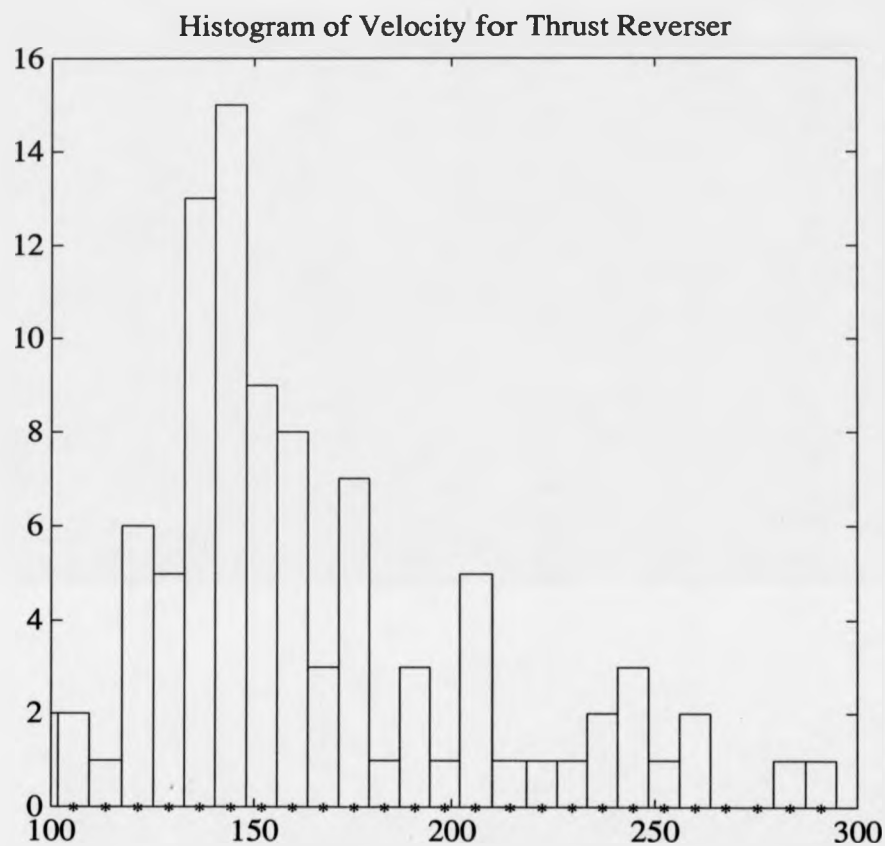


Figure 6.35: PIDV Histogram of Velocity for Thrust Reverser

Mean Velocity = 63.2 m/s , s.d. = 8.9 m/s
Mean Vector Angle = 89.5 deg, s.d. = 7.4 deg
Mean Particle Size = 14.2 pixels, s.d. = 9.9 pixels
Mean Particle Displacement = 15.5 pixels, s.d. = 2.2 pixels

Table 6.5: Summary of Spatial Pairing Analysis Result for Example ARA Test

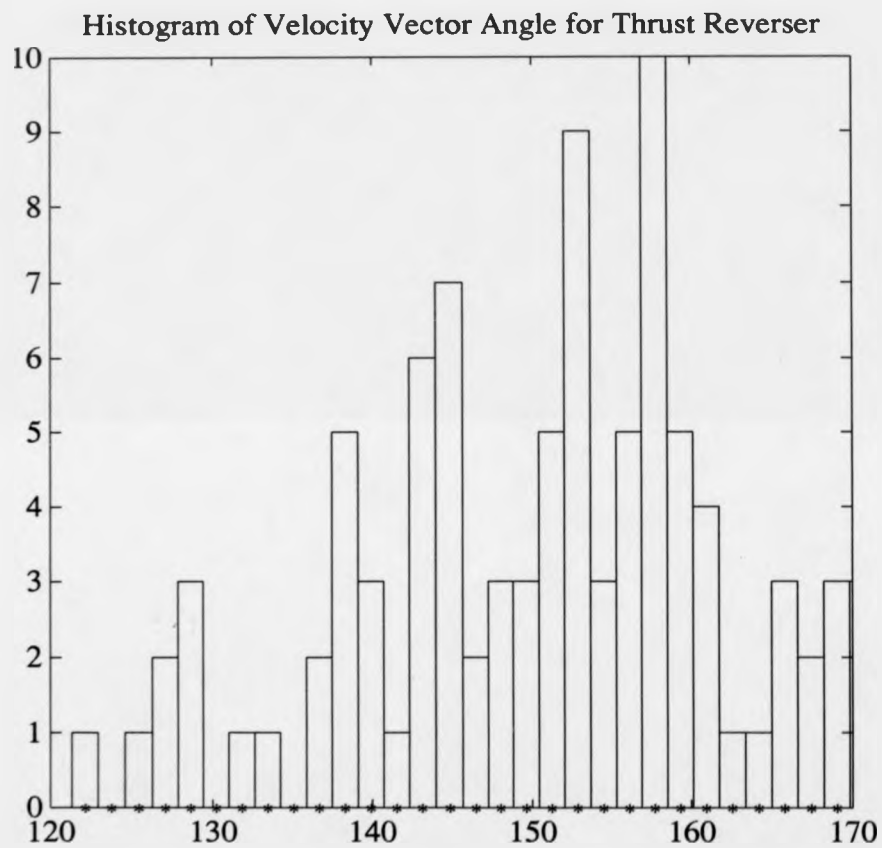


Figure 6.36: PIDV Histogram of Velocity Vector Angle for Thrust Reverser



Figure 6.37: Scan of PIDV Print, Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees

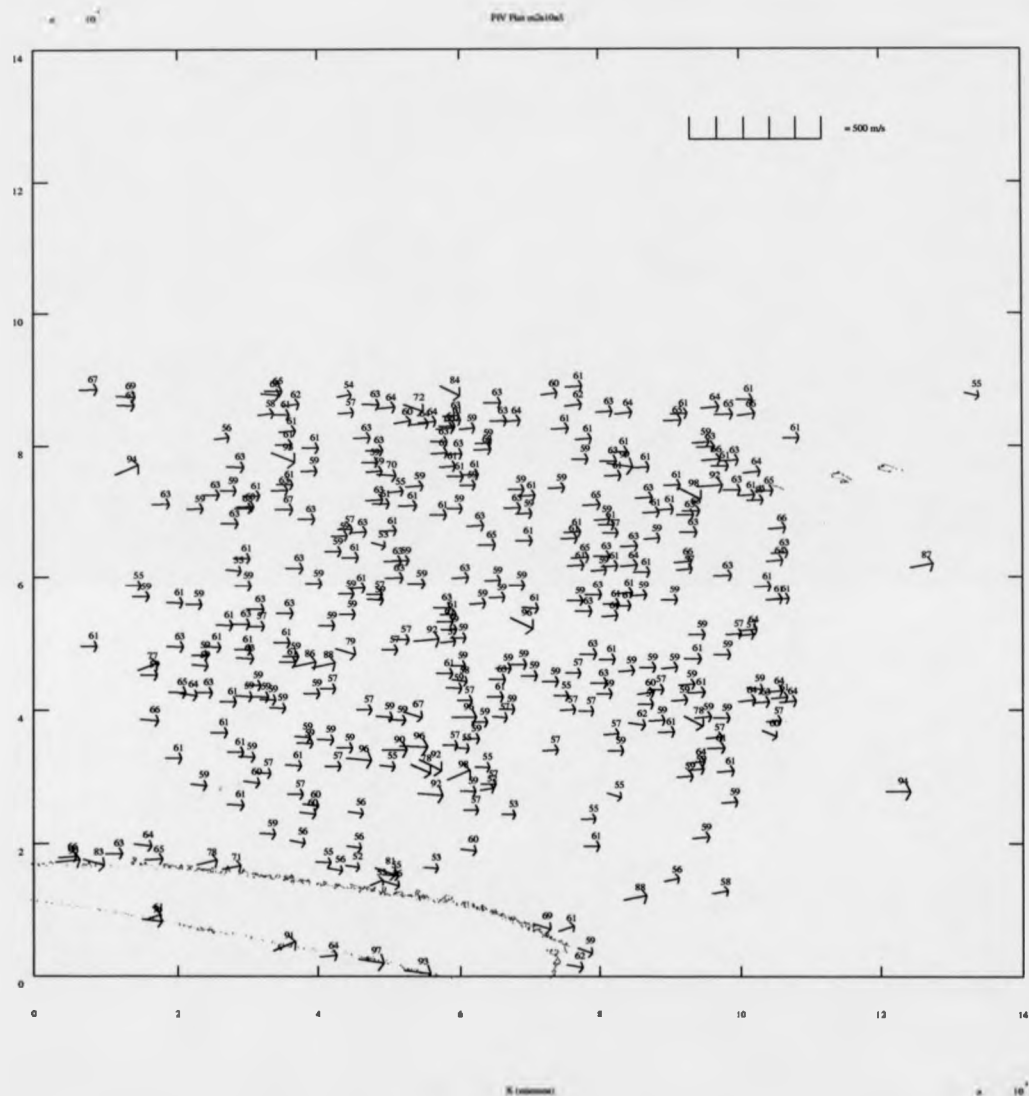


Figure 6.38: PIDV Vector Plot Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees

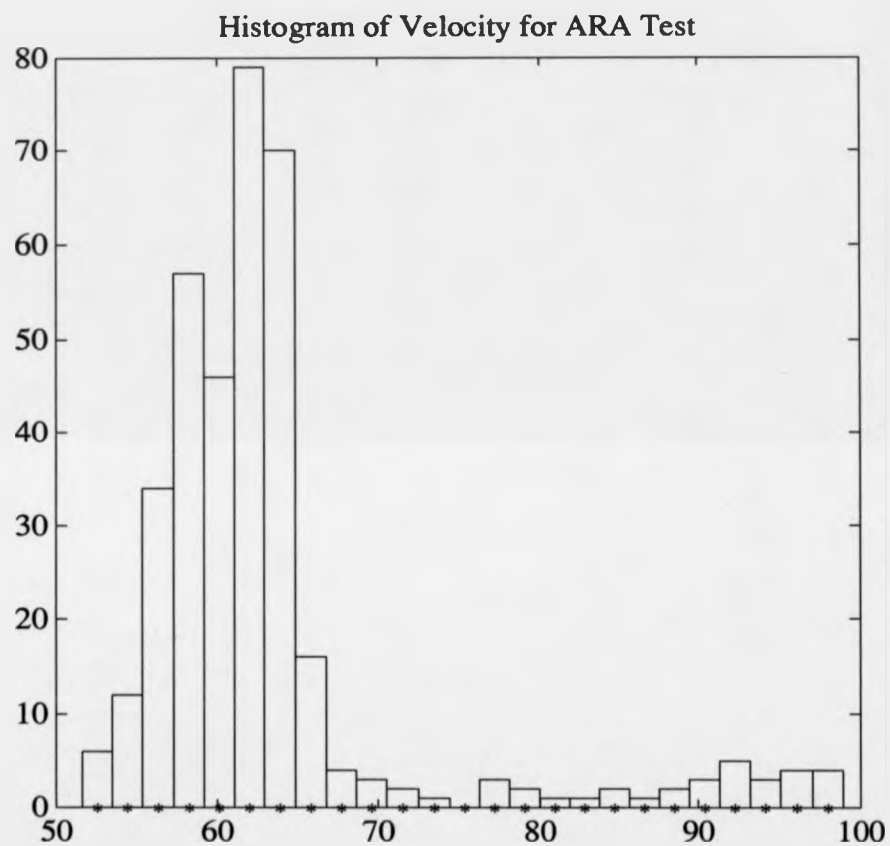


Figure 6.39: PIDV Histogram of Velocity, Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees

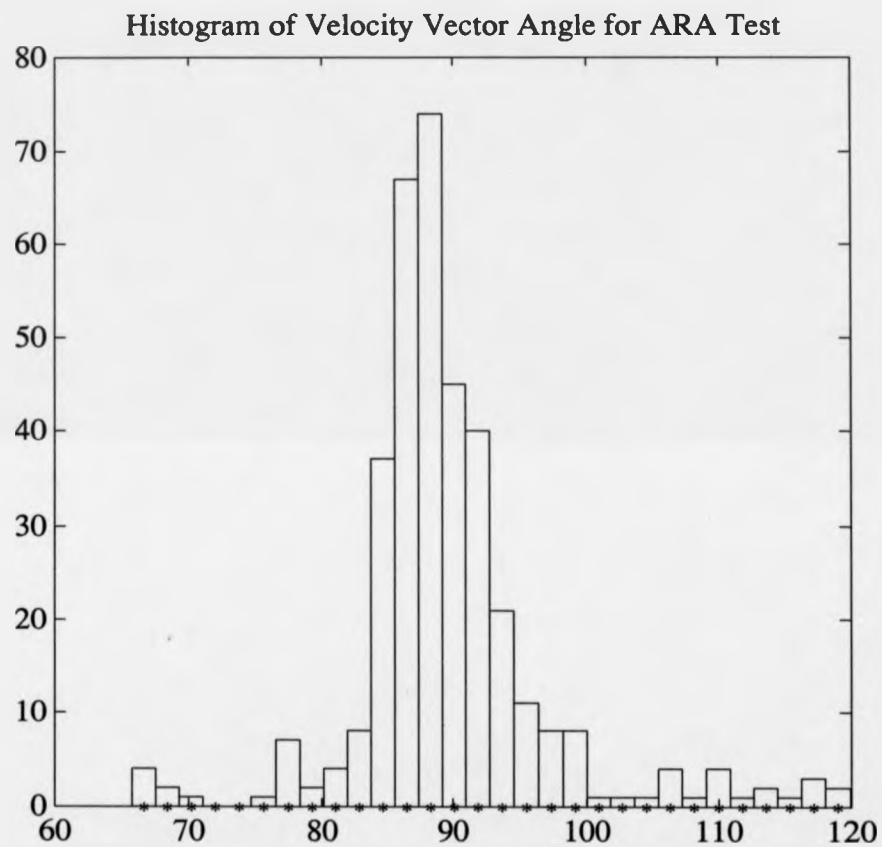


Figure 6.40: PIDV Histogram of Velocity Vector Angle, Mach No. 0.2, Frame No. 7, 10 Microsecond Pulse Separation, Model at Incidence of 5 Degrees

6.11 Conclusion

This chapter has explored the analysis of PIDV data by Autocorrelation and a Spatial Pairing method. It has been demonstrated that both techniques have their particular advantages. These are summarised below.

Advantages of the 2D correlation technique,

- i) Gives a clear visual indication if a number of differing flow vectors are present in a given area.
- ii) It can be implemented using dedicated FFT hardware, which is widely available.
- iii) The method is based on standard mathematical techniques.
- iv) The method can be spatially optimised, but specialised hardware is not then available.

Disadvantages of the 2D correlation technique,

- i) Velocity vectors are not easily traceable to the specific 'particles' in the flow image which generated them.
- ii) The method is very time consuming if implemented in the Fourier domain without dedicated hardware, or high powered processing.
- iii) Each correlation must take place over a small area. If the area considered is too large, then it becomes difficult to extract correlation peaks. If a complete vortex were contained in the area considered, for example, then this would produce many peaks in a ring.

Advantages of the spatial pairing analysis technique,

- i) Analysis is rapid for sparse fields, without dedicated hardware.
- ii) The computed velocities are easily matched with the pairs which gave rise to them. This permits a better understanding of flow structures to be obtained in sparse data fields.

- iii) As the technique does not require dedicated hardware it is easily transferred between installations.
- iv) Simple statistics may be performed upon the relatively large number of velocity data points obtained. For example, mean, standard deviation and the velocity distribution are all easily obtained.

Disadvantages of the spatial pairing analysis technique are,

- i) It becomes rather slow when very dense fields are considered.
- ii) Dedicated hardware is not available to speed the technique.
- iii) Prominent flow directions are not as well highlighted as with the Autocorrelation method.

Bibliography

- [1] P. J. Bryanston-Cross, "ARA Final LINK report: Review of the scientific aspects of the project", 1991.
- [2] "Particle Image Displacement Velocimetry", Lecture Series 1988-06, von Karman Institute for Fluid Dynamics, Chaussee de Waterloo 72, B - 1640 Rhode Saint Genese, Belgium, March 21-25, 1988.
- [3] Borne & Wolfe, Principles of Optics, Pergamon Press, 1959.
- [4] P. J. Bryanston-Cross, A. H. Epstein, "The Application of Submicron Particle Visualisation for PIV (Particle Image Velocimetry) at Transonic speeds. ", Progress in Aerospace Science, pp.1-31, Oct, 1990.
- [5] P. J. Bryanston-Cross, S. P. Harasgama, C. E. Towers, T. R. Judge, D. P. Towers, S. T. Hopwood, "The Application of Particle Image Velocimetry (PIV) in a Short Duration Transonic Annular Turbine Cascade", ASME 1991.
- [6] I. Grant, G. H. Smith, "Modern Developments in Particle Image Velocimetry", Optics and Lasers in Engineering, Vol. 9, pp. 245-264, 1988.
- [7] R. Grousson, S. Mallick, "Study of Flow Patterns in a Fluid by Scattered Laser Light", Applied Optics, Vol. 16, pp. 2334, 1977.
- [8] D. B. Barker, M. E. Fourney, "Measuring Fluid Velocities with Speckle Patterns", Optics Letters, Vol. 1, pp. 135, 1977.
- [9] T. D. Dudderar, P. G. Simpkins, "Laser Speckle Photography in a Fluid Medium", Nature, Vol. 270, pp. 45, 1977.

- [10] R. J. Adrian, "Scattering Particle Characteristics and their Effect on Pulsed Laser Measurements of Fluid Flow: Speckle Velocimetry vs Particle Image Velocimetry", *Applied Optics*, Vol. 23, No. 11, pp. 1690-1691, 1984.
- [11] R. J. Adrian, C. S. Yao, "Development of Pulsed Laser Velocimetry for Measurement of Turbulent Flow", In *Proceedings of the Eighth Biennial Symposium on Turbulence*, ed. G. Patterson, J. L. Zakins, University of Missouri-Rolla, pp. 170-185, 1983.
- [12] R. J. Adrian, C. S. Yao, "Pulsed Laser Technique Application to Liquid and Gaseous Flows and the Scattering Power of Seed Materials", *Applied Optics*, Vol. 24, pp. 44-52, 1985.
- [13] R. Meynart, "Speckle Velocimetry Study of a Vortex Pairing in a Low-Re Unexcited Jet", *Phys. Fluids*, Vol. 26 No. 8, pp. 2074-2079 1983.
- [14] L. M. Lourenco, M. C. Whiffen, "Laser Speckle Methods in Fluid Dynamics Applications", In *Proceedings of the 2nd International Symposium on the Application of Laser Anemometry to Fluid Mechanics Methods in Fluid Dynamics Applications*", ed. D. F. Durao, Instituto Superior Technico, Lisbon, Portugal, Paper 6.3, 1985.
- [15] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in C The Art of Scientific Computing*, Cambridge University Press, ISBN 0-521-35465-X, 1988.
- [16] N. M. Myrman, "Processing Fluid-flow Data the Easy Way", *Lasers & Optonics*, pp. 58-60, 1990.

Chapter 7

Conclusion

A summary and discussion of the main points from the work are given below.

The first five chapters of the thesis have concerned fringe analysis.

A general and robust two dimensional phase unwrapping technique has been presented. The technique is particularly aimed at addressing the problems posed by natural or aliasing induced discontinuities. Phase unwrapping approaches have not fully addressed such problems in the past. The success of the technique has been demonstrated in a wide variety of fringe analysis applications. These applications have included vibration analysis, deformation measurement and flow visualisation. The technique has been shown to be applicable to wrapped phase maps generated by the FFT and Phase Stepping methods (the most prominent methods for the generation of wrapped phase maps). The technique may be applied in any application which yields wrapped phase maps, for example Satellite Radar Interferometry or Moire.

The phase unwrapping approach has a clear structure. It makes use of a regional and pixel level. Information across a range of scales in the image is therefore combined to improve the confidence of phase unwrapping. This structure provides scope for enhancement with future developments. For example, the weighting functions may be modified to take account of any specialised discontinuity types, which might be found in future applications.

Interest in the subject of fringe analysis has increased over the last decade. This has been due, in part, to developments in Video technology and Computer Engineering which have encouraged researchers to experiment. It is always desirable to automate analysis systems, to reduce the need for expert

knowledge.

Interferometry is used in a range of applications where automation would be welcome:

- i) Deformation measurement
- ii) Assessing modes of vibration.
- iii) Stress / Strain analysis.
- iv) Flow visualisation.
- v) The detection of delaminations, non-adherence and cracks.

The question arises of how far automatic methods can proceed in fringe analysis, especially in relation to the special problems found in fringe fields. The traditional technique of Fringe Tracking has clear limitations. The direction of motion is not coded and the analysis is not reliable where the field is discontinuous.

Phase Stepping and FFT Methods permit automatic coding of elevation and depression. The Phase Stepping method is prone to non-linearities in the response of the detector, as the phase is computed from a series of intensity measurements. The FFT method relies on the deviation in straightness of a set of fringes. The accuracy of the technique is more reliant on the spatial measurement of fringe position, rather than intensity.

The FFT method requires an automatic strategy for extraction of the side lobe. In order to perform this task reliably the carrier frequency must be much higher than any component in the original signal. This requires careful experimental design. Such design issues have been addressed in the work. The work has illustrated a method of isolating the side lobe by a Fourier windowing technique. It has been shown that this method can introduce phase errors, if the entire signal is not within the window. An alternative method is to compute the Fourier transform of a background image, and subtract this from the Fourier transform of the interferogram. This has been illustrated with reference to work at BAe.

The phase unwrapping approach is required to decode the interferogram and supply a result in terms of the parameter measured.

One of the points highlighted in the work has been the importance of the successful detection of fringe edges at 2π phase jumps. This is sometimes overlooked and implemented in an ad hoc manner. It has been shown that adaptive thresholding provides one method of improving the performance of edge detection in fringe analysis, particularly when applied to the widely used Sobel edge detection technique.

It has been shown that a high degree of automation may be achieved by the combination of regional and pixel level unwrapping strategies.

Pixel level unwrapping strategies are typified by the Cellular Automata technique, Minimum Spanning Tree and Cut Methods. Any of these may be applied in conjunction with a higher level regional strategy to improve confidence in the solution obtained. The Graph and weighting factors developed enable the consistency of the field to be mapped.

A tile level of phase unwrapping has been described which permits the isolation of large scale discontinuities, typified by shadows or aliasing problems. It has been shown that the size of tiles employed should be related to the size of the discontinuities which are to be isolated.

In computing the tile level weightings, it has been shown that the strategy adapts the significance of the various factors to match the field. That is, the factors are normalised based upon their distribution in the particular image being processed, rather than having a predefined absolute range from bad to good.

To produce a system capable of accurate measurement all steps from generation of the interference pattern to delivery of the final unwrapped solution must be carefully considered to evaluate the errors introduced. The function of the computer in such systems is to convert data in the form of the interferogram into other more accessible forms whilst introducing as few distortions as possible. The computer forms just one link in the chain.

This reflection prompted an investigation of the CCD camera and the digitisation process, the component of the system which directly supplies input to the computer. The response of the device capturing the interferogram is an important factor in the system. It was seen that the gain setting of the digitising board effected the noise distribution. It was seen that the signal noise level can represent an intensity variation, from frame to captured

frame, of as much as + or - 8% for some values of gain. If the camera digitiser combination is calibrated the noise level may be reduced by summing frames. The phase stepping method is more sensitive to the response of the camera/digitiser combination than the FFT method.

Devices for digitising imagery continue to increase in both spatial resolution and intensity resolution. The fringe analysis system developed in this work has been designed with this in mind. The package is not limited by image size, and is designed to deal with up to 16 bits of intensity resolution.

Parallel approaches are well matched to the problem posed by increasing image size. Parallel implementation of the MSTT phase unwrapping strategy has been considered. It has been shown that the tiling approach is well suited to parallelisation as each tile is solved independently of its neighbours, which means tile solutions may be computed in parallel. The parallel Minimum Spanning Tree algorithm has been described, with a time complexity of $O(\log^2 n)$ time on $\frac{n^2}{\log^2 n}$ processors, where n is the number of pixels or tiles depending upon the level considered. The sequential algorithm has a time complexity of $O(n^2)$. It has been shown that implementation of the phase unwrapping strategy on a Gated Connection Network, in parallel, is possible. This is an SIMD processor array optimized for MST computation (and similar problems).

Chapter 6 discusses Particle Image Displacement Velocimetry. There is a need to provide sophisticated aerodynamic diagnostics in order to encourage aircraft manufacturers to return to UK testing facilities such as ARA Bedford.

PIDV is a very important technique. It permits the visualisation of turbulent flow, without averaging effects, and gives quantitative data.

PIDV is undergoing rapid development. Advances in video technology will effect the way PIDV is applied. The next step will be to apply the technique with high resolution CCD arrays.

It was seen in Chapter 6 that a number of PIDV processing options exist. The spatial pairing strategy has been developed with the requirements of Transonic Video PIDV in mind. The data points are sparsely distributed and require online processing. It is hoped that, with suitable processing capacity, such results could be displayed in real time during aerodynamic

testing providing vast amounts of quantitative data, but also in order that attention might quickly be directed to the flow domains of significance. Laser doppler anemometry by contrast samples one point at a time and therefore produces an average impression of the flow field.

In contrast to this development of Video PIDV, lies the potential of Holographic PIDV. That is particle images stored three dimensionally in a holographic medium. It has recently been shown that fringe patterns encoding velocity and direction information may be generated from double pulse holograms, recording the 3D spatial positions of particles during the first and second pulse. This should provide much higher resolution results. æ

Appendix A

Documentation For FRANSYS Fringe Analysis System

A.1 Introduction

FRANSYS is a software package which permits automatic fringe analysis. The package supports both Fourier Transform (FT) and Phase Step fringe analysis. FT analysis is performed according to the method described by Takeda [1]. The Phase Stepping procedure uses 3 interferograms, as described by Dandliker [2]. The package implements an advanced automatic phase unwrapping algorithm by Judge [3, 4].

A useful review of Fringe Analysis techniques is given by Reid up to 1986/87 [5].

FRANSYS has been conceived as a portable package. It has been extensively tested on both PC compatibles and Sun Workstations. The main analysis package is identical on both systems. However, it has been found useful to supply an additional graphical interface for the Sun System to compliment the windowing environment of Sunview or Open Windows.

Throughout the documentation a computer generated FT image is used as an example, see Fig A.1. This has been generated from the module 'mkfft' described within this document. The test image represents a concentric fringe pattern modulated by a set of carrier fringes. The carrier frequency has been specified as 0.25 cycles/pixel (that is one carrier fringe is four pixels wide) and the concentric fringe frequency as 0.025 cycles/pixel. The image is 256

by 256 pixels.

The package consists of a series of modules which will be described in the sections which follow.

A.2 Sunview Window Interface

This interface is available with the Sun version of the software.

The sunview windowing environment may be invoked by typing

sunview

The graphical interface to FRANSYS is called 'fw'. This may be started by typing

fw&

within one of the windows of sunview. Figure A.2 shows an example of the screen display.

The main fran package may be executed from a button selection within this environment. The interface incorporates an image viewer which operates on 8 or 16 bit per pixel TIF (Tagged Image Format) files. A zoom function is supplied for enlarging detail.

There are facilities to generate Postscript from the images, print directly to a specified laser printer and to convert images into the Sun Raster file format.

Project files, which record analysis options for specific images, may be created and edited under the menu system.

A.2.1 TIF Image File Display

In order to display an image file, enter the file name in the field labeled 'Image File Name: '. This field is situated near the top of the window interface. In order to enter text, the mouse cursor should be positioned just beyond the colon and the left mouse button pressed. The file 'test.tif' may be used as an example. This is the image of Fig A.1. When the file name has been correctly entered, 'press' the button labeled 'Read Image File' (a 'button' icon is 'pressed' by moving the mouse cursor to it and pressing the left button on the mouse).

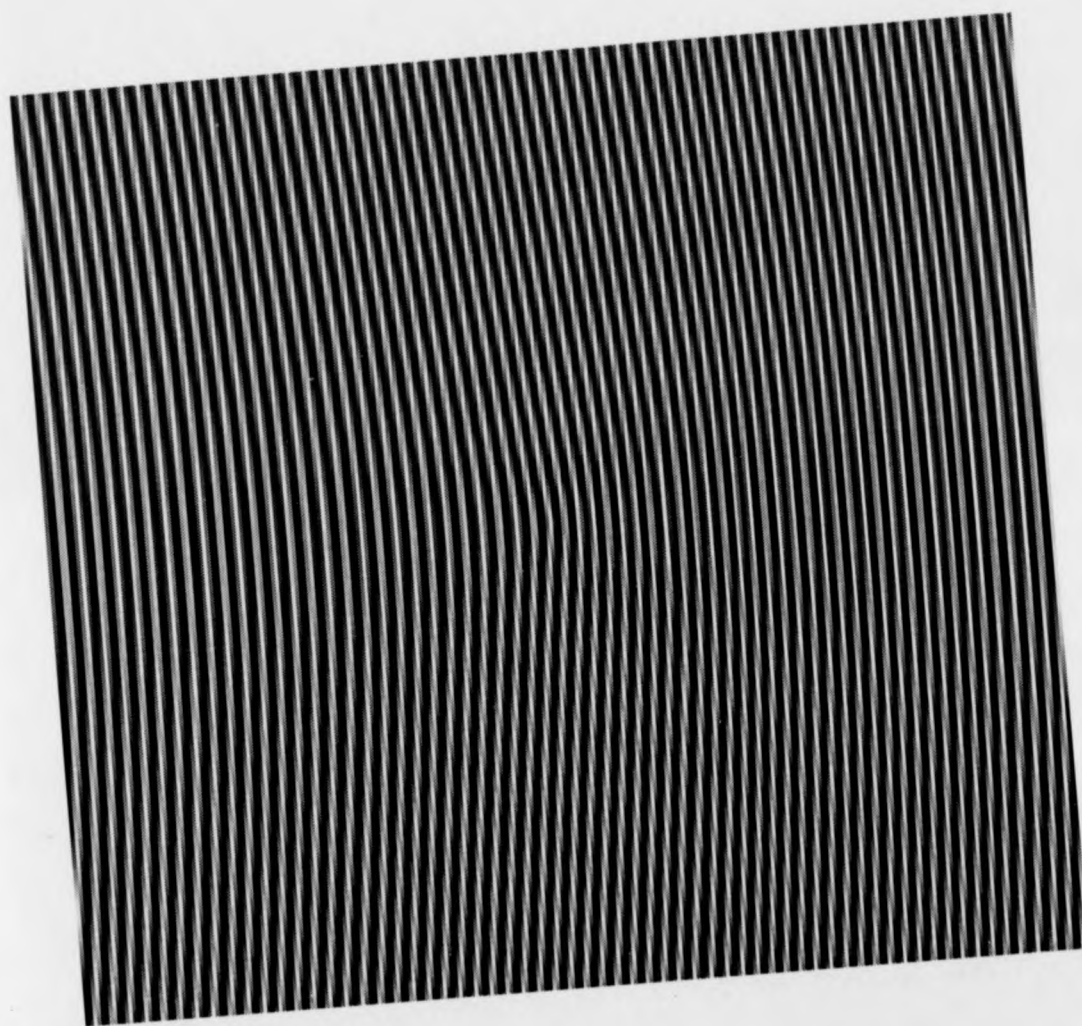


Figure A.1: Example FFT Image Generated By mkfft

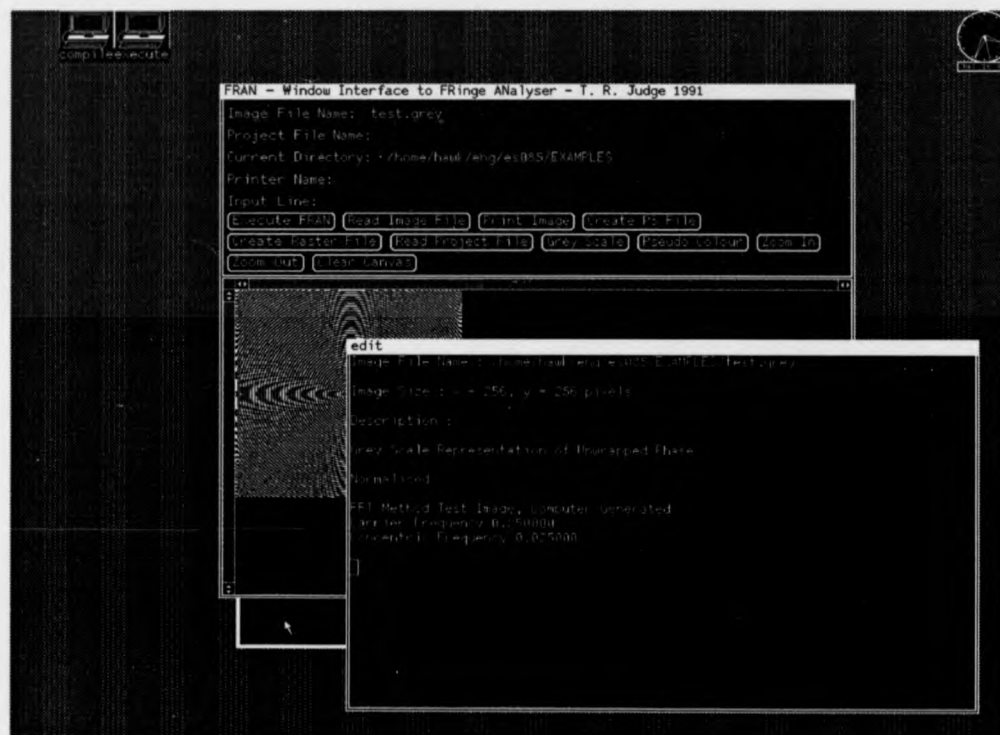


Figure A.2: Sunview Window Interface to FRANSYS

Assuming all is well, the image will appear in the canvas area of the interface. The window may be resized if the whole image is not visible. If the image is too small 'press' the button labeled 'Zoom In'. If the image is too big 'press' the button labeled 'Zoom Out'.

Information about the image is displayed within the window which started the 'fw' process. This information includes the size of the image in pixels and a textual description of the image.

The -f option of the 'mktif' command may be used to specify an image description.

The details of the example image as displayed in the window are shown in Figure A.3.

Image File Name : test.tif
Image Size : x = 256, y = 256 pixels
Description :
FFT Method Test Image, Computer Generated
Carrier Frequency 0.250000
Concentric Frequency 0.025000

Figure A.3: Image Data Display in Sunview

A.2.2 Pseudo Colour

To view an image in Pseudo Colour press the 'Pseudo Colour' button. This gives an increase in visual dynamic range over a grey scale image and more detail is then visible. High intensity points are shown as yellow, middle intensity as red, and low intensity as blue. Zero intensity is shown as black.

A.2.3 Grey Scale

To switch to a grey scale image display simply press the button labeled 'Grey Scale'.

A.2.4 Printing an Image File

If the image is to be inserted in a document then it is worthwhile discovering whether the document preparation system being employed accepts TIF files

(many do, for example Ventura). If this is the case then the original TIF image files may be used as they are. The next option is to generate Encapsulated Postscript files for the images and investigate how to insert them into the document. Most document preparation systems allow insertion of Postscript. In order to generate a Postscript file simply press the button labeled 'Create PS File'. The source image file is that indicated by the 'Image File Name :' field. The image does not necessarily have to be on display. The file generated is given the same name as the original image file, with the extension '.ps' appended.

To print an image directly to a printer, first specify the name of the printer in the 'Printer Name :' field. In commands such as 'lpr -Pxx', xx is the printer name. This should be the name of a Postscript printer. When this has been correctly entered simply press the button labeled 'Print Image'. The button will turn dark grey while the various files are being generated. When it returns to its original colour the image will be in the printer queue.

A.2.5 Reading a Project File

The project file contains various options and variables to do with fringe analysis. It is described in detail later on. A project file is read in a similar manner to an image. The project file name is specified adjacent to the 'Project File Name:' field and read by pressing the button 'Read Project File'. Alternatively a project file may be loaded as 'fw' is started, using the syntax

```
fw <project file> &
```

Try this, for example, with the project file for the test image 'test.cfg'. To view the data contained in the project file, move the mouse cursor so that it is over the canvas area where images are displayed. Hold down the right button. A menu will then appear. This menu will contain the project file data which may be amended as required.

A.2.6 Creating a Project File

In order to create a project file specify a name next to the field 'Project File Name :'. Next move the mouse cursor so that it is over the canvas area.

Hold down the right button and a menu will appear. This menu will contain default information about the type of analysis to be performed. It is similar to the project file itself (see below).

Phase Step analysis is assumed by default. To change this, simply select the option labeled Phase Step and release the mouse button. The menu will then disappear. To see the change in the menu, hold down the right button once again. The mode of analysis should then have changed to FFT. This is the general way to change an option which has several modes.

To change an option that requires numeric or textual input, enter the data (numeric or textual) adjacent to the field 'Input Line :'. Move the cursor again over the canvas and select the menu by holding down the right button. Selecting an option will then cause input to be taken from the 'Input Line :*' field.

The FRAN package is run by pressing the button labeled 'Execute FRAN'. The project data is saved to the project file just before execution commences.

A.2.7 Converting To Sun Raster File Format

Set up the TIF image file name in the 'Image File Name:' field, then press the 'Create Raster File' button. This will create a raster file of the same name as the original image with the extension '.ras' appended. The original image is not deleted.

A.2.8 Removing an Image From the Canvas

To remove an image from the canvas press the button 'Clear Canvas'.

A.3 The FRAN Program

The main analysis program is called 'fran'. It has a window based interface (described above), but may also be used from the command line. The syntax is then

```
fran <project file>
```

The project file is an ascii text file. The options it contains control the 'fran' program. The format of the project file and the available options are

described below.

A.3.1 Project File Format

Each line in the project file is given to a separate option. The file begins by defining the mode of analysis, either by phase stepping or the FFT approach, and ends by specifying the images over which the analysis should be performed. Other options effect the data saved during processing, wrapped phase map, edge detection etc. In order to function as a project file each line of the file should contain an option, in the order shown below.

- i) PHASE_STEP [phase step] or FFT [carrier freq] or FFT_USING [raster n] or READY_WRAPPED

The PHASE_STEP option is used to indicate that quasi-heterodyne analysis is required. The argument [phase step] should be an angle in degrees. Three input interferograms are used. In this case, with three fringe fields at a phase step of α , the phase ϕ at a given pixel in the wrapped phase map, may be calculated [2] from

$$\phi = \arctan \left[\frac{(I_3 - I_2) \cos \alpha + (I_1 - I_3) \cos 2\alpha + (I_2 - I_1) \cos 3\alpha}{(I_3 - I_2) \sin \alpha + (I_1 - I_3) \sin 2\alpha + (I_2 - I_1) \sin 3\alpha} \right] \quad (\text{A.1})$$

where I_1 , I_2 and I_3 are the intensities of the interferograms at the three phase positions α , 2α and 3α respectively.

For FFT analysis the option FFT is employed. This has two forms. The first FFT_USING is followed by a raster number. The carrier frequency, in this case, is extracted from the specified scan line. The selection is made by examining the power spectrum of the indicated scan line for the frequency with the greatest power. In the second form, FFT [carrier freq], the carrier frequency is specifically set in cycles per pixel.

The READY_WRAPPED option should be used if the wrapped phase is computed by some external process. The input file, in this case, should consist of a list of single precision floating point

numbers in their binary format (that is 4 bytes per pixel value), representing the wrapped phase values between $-\pi$ and π . The size of the image is passed separately into a file of the same name as the input file with the extension '.fps', this should contain two integers in their binary format (32 bits per integer on Sun Machines), specifying the x and y resolution of the image respectively. In order to check whether the wrapped phase map has been interpreted correctly, it is written to a TIF output file with extension '.tan' (if the SAVE_TAN_ON option is enabled).

ii) TILE_SIZE [n]

Phase unwrapping takes place over a grid. That is, the field is divided into tiles which are unwrapped separately. These are then recombined according to a confidence tree. The option TILE_SIZE specifies the size of the TILE in the grid. There is an overlap of 4 pixels between tiles which should be included in the tile size. The minimum tile size is 6 pixels.

iii) BLUR [n] or MEDIAN [n]

BLUR [n] specifies the number of iterations for a pre-processing averaging filter. This filter averages by replacing the centre pixel by the average of a 3×3 pixel area. This option is normally used with speckle fringe fields. The filtered image(s) are written to files with the same prefix as the original images but extension .prp. The number of bits is expanded to 16 per pixel.

MEDIAN [n] specifies the number of iterations for a pre-processing Median filter. This filter replaces the centre pixel by the median of a 3×3 pixel area. This option is normally used with speckle fringe fields. The filtered image(s) are written to files with the same prefix as the original images but extension .prp.

iv) MOD_PERCENT [n] (where n is a floating point number)

Low modulation is an indicator of bad data. This option specifies an area threshold for low modulation points. If the area of low

modulation exceeds this threshold the tile is deleted from the solution. The area threshold is specified as a percentage of the tile area. A value of 10 % has been found to be effective.

v) NORMALISE_ON / OFF

This option indicates whether the input image(s) should be normalised. Normalisation stretches the grey scale range of the image to cover the available range. The original grey scale range of the image is displayed. The normalised images are written to files with the same prefix as the original images but extension .prp.

vi) SAVE_TAN_ON / OFF

This option specifies whether an image containing the wrapped phase map should be saved. If an image is generated it is given the extension '.tan'. The example of Fig A.4 shows the wrapped map of the test image.

vii) SAVE_EDGES_ON / OFF

This option specifies whether an image containing the Sobel edge detection of the wrapped phase map should be saved. If an image is generated it is given the extension '.edg'. The example of Fig A.5 shows the edge detection of the wrapped map of the test image, with the tree added. Edges detected above the high threshold are shown in solid black, those above the lower threshold in grey.

viii) SAVE_LOW_MOD_ON / OFF

This option specifies whether an image containing the low modulation noise in the wrapped phase map should be saved. If an image is generated it is given the extension '.mod'.

ix) SAVE_GREY_ON / OFF

This option specifies whether an image containing the normalised unwrapped phase map should be saved. If an image is generated it is given the extension '.grey'. The example of Fig A.6 shows the unwrapped map of the test image.

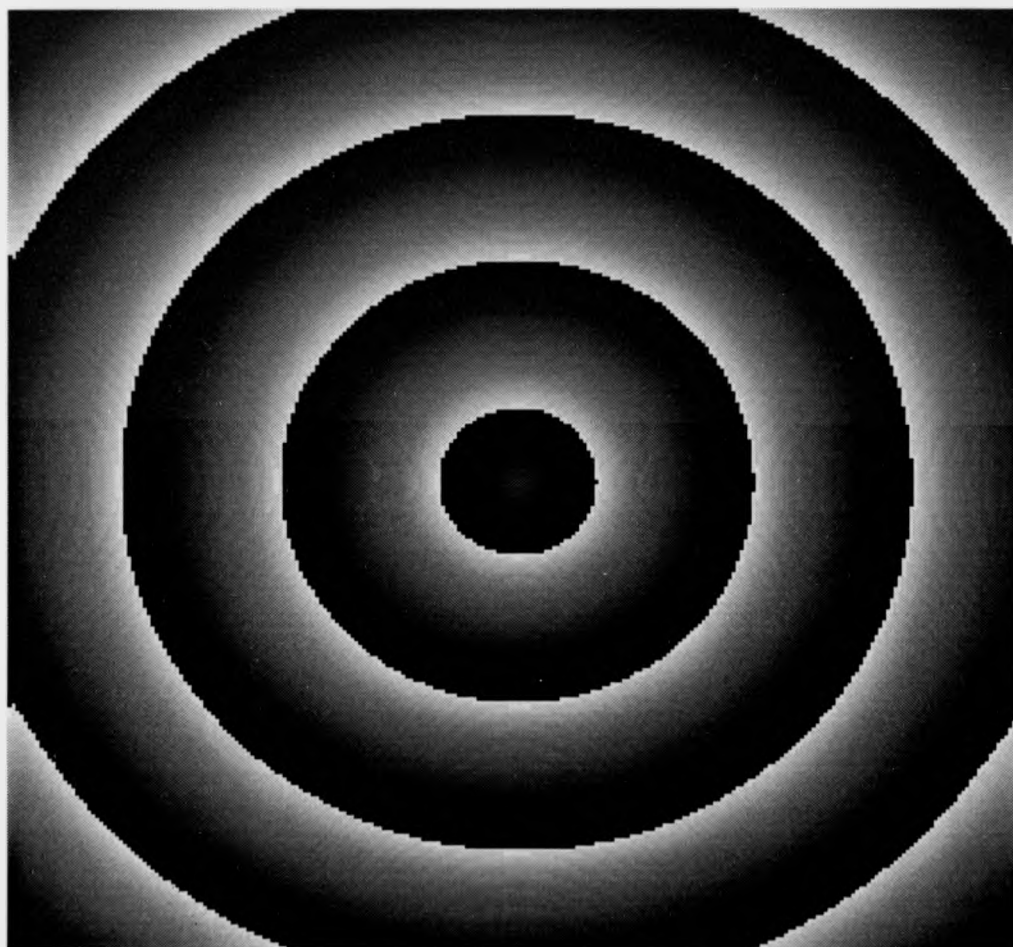


Figure A.4: Example FFT Wrapped Map

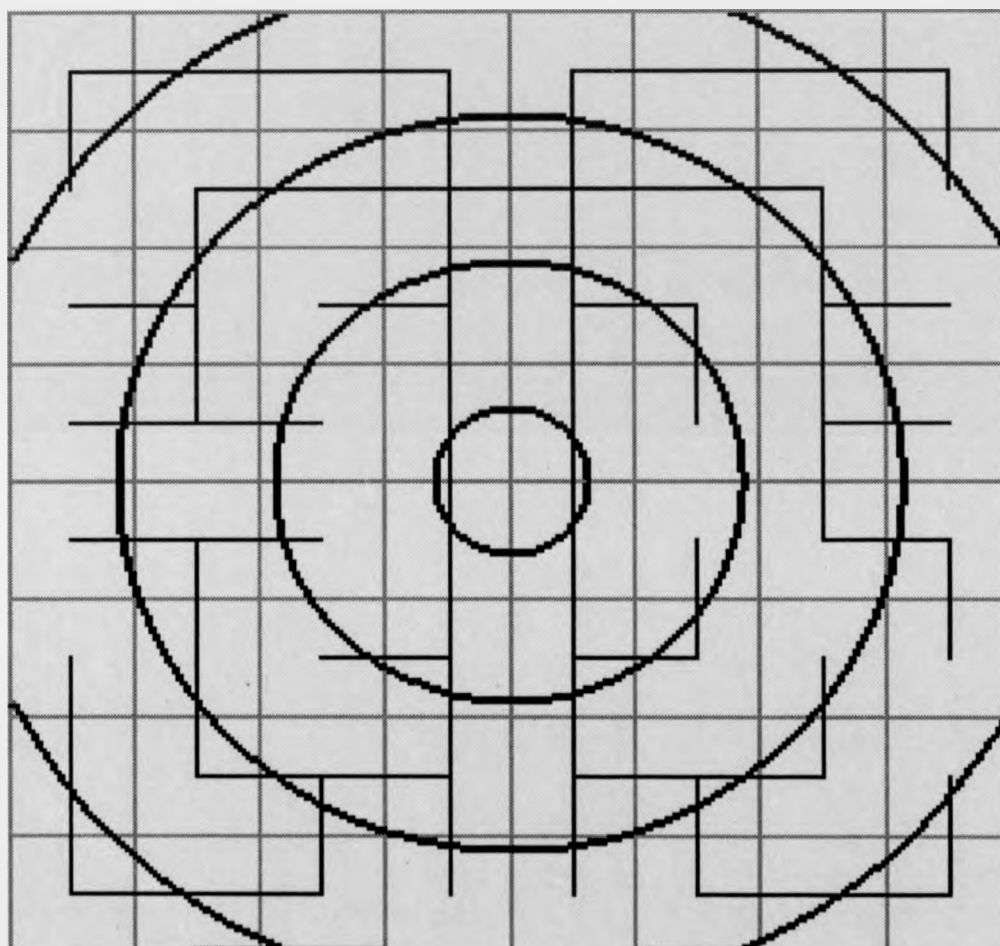


Figure A.5: Example FFT Edge Detection of Wrapped Map, With Tree Added

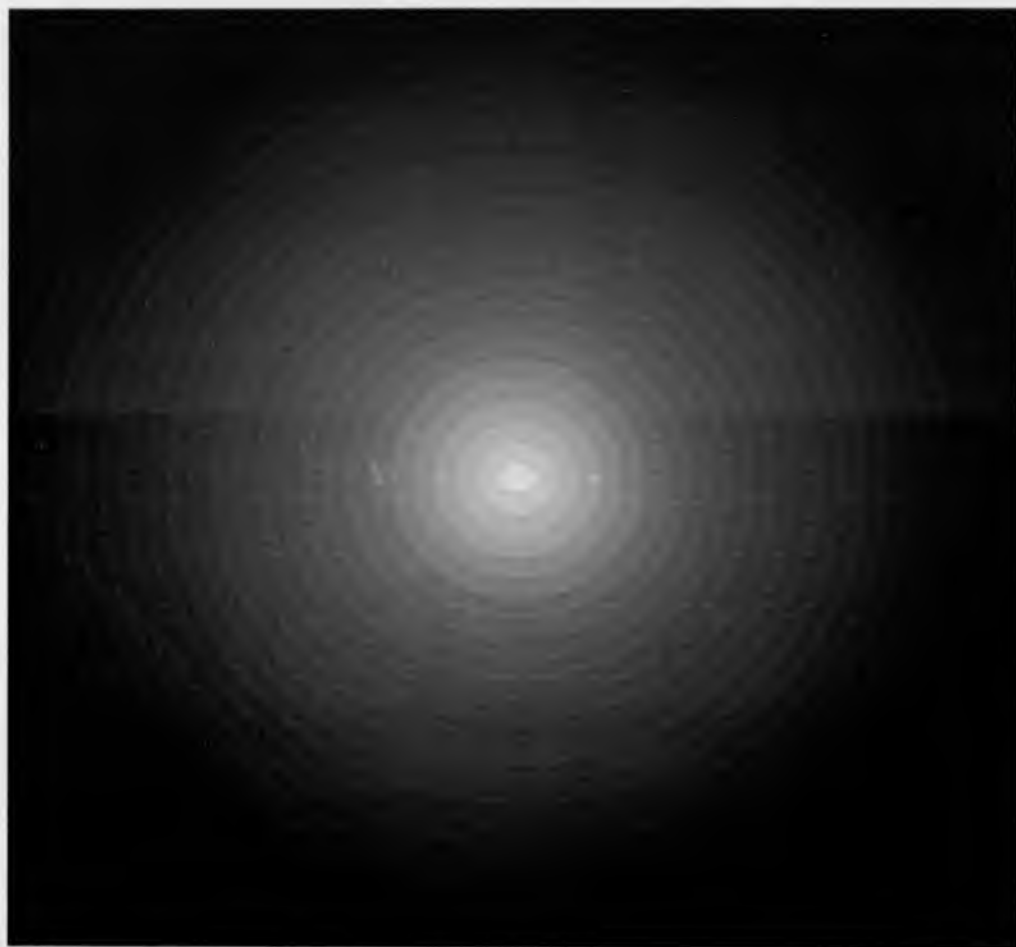


Figure A.6: Example FFT Unwrapped Map

x) SAVE_TREE_ON / OFF

This option specifies whether the Minimum Spanning Tree used to unwrap the field should be saved. The file created is given the extension '.tree'. This file is not an image but a binary data file containing a list of tile coordinates and interconnection weights. If the SAVE_EDGES option is set, ON, then the connection tree is drawn onto the edge detected image with extension '.edg', as shown in Fig A.5.

xi) SAVE_FFTS [start scan],[end scan] or SAVE_FFTS_OFF

This option is specific to fringe analysis by the FFT method. It indicates that spectral data about the raster scan lines between the rasters [start scan] and [end scan] should be saved. Five file types are created. The file names are generated automatically with the following format

< prefix >.< file type key letter >< raster number
>.dat

The following is a list of the file type key letters and what the files contain. As an example data concerning raster 128, the central raster of the test image, has been saved. This data is shown in several plots below.

r) This is the original intensity data for the raster, Fig A.7.

w) This is the original intensity data for the raster weighted by a papoulis window, Fig A.8.

f) This is the power spectrum of the specified raster, shown in Figure A.9 by a solid line.

x) This is the papoulis window employed to weight the power spectrum of the specified raster, shown in Figure A.9 by a dash-dot line. The data file is zero padded outside of the weighted area so that it aligns with the data file generated by f) above.

t) This is the power spectrum of the translated side lobe (after weighting by the papoulis window), shown by the solid line in Figure A.10.

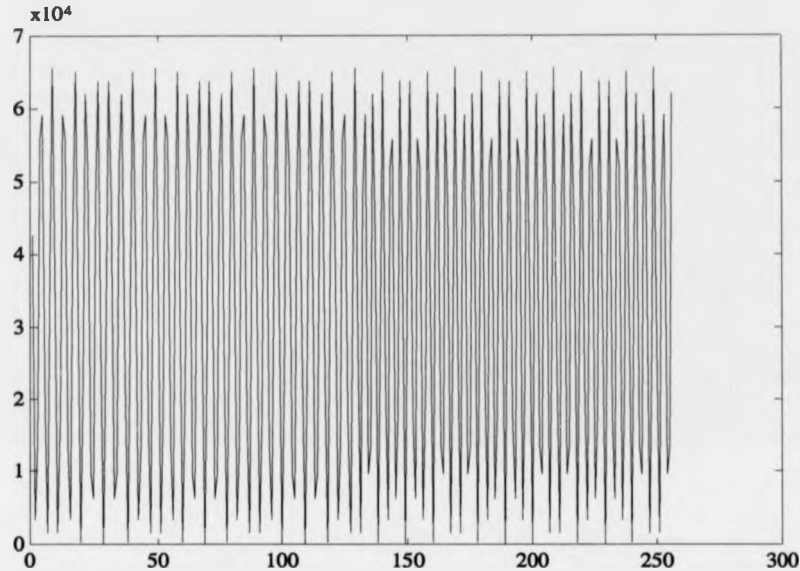


Figure A.7: Original Intensity Data For Raster 128

xii) SINGLE_TAN or DOUBLE_TAN

This option is included to allow doubling of the number of fringes appearing in the wrapped phase map. This is achieved by using the 'atan' function rather than the 'atan2' function to compute principal values.

The edges of fringes in the wrapped phase map are used as one measure of confidence. In some cases it is possible to improve the unwrapped solution by doubling the number of fringe edges.

xiii) RAW_DATA or FRINGE_COUNT or POLY_CORRECT n or POLY_SMOOTH n

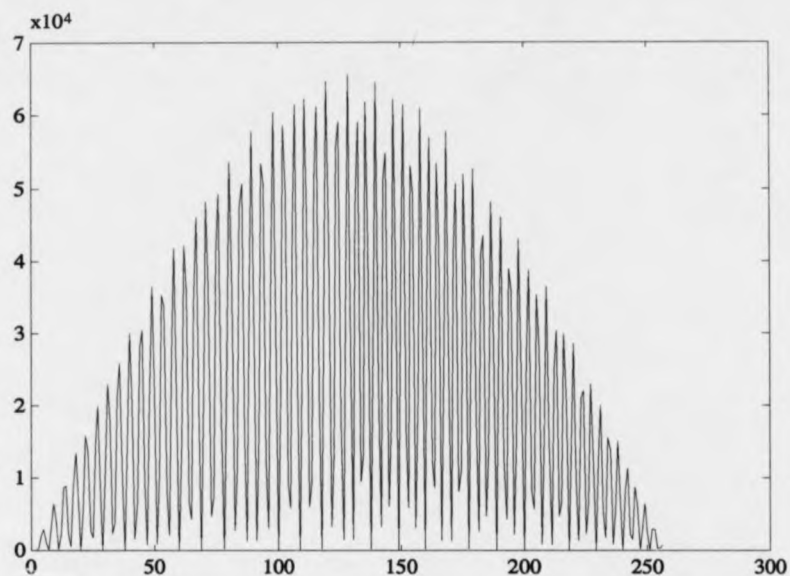


Figure A.8: Intensity Data of Raster 128 Weighted by Papoulis Window

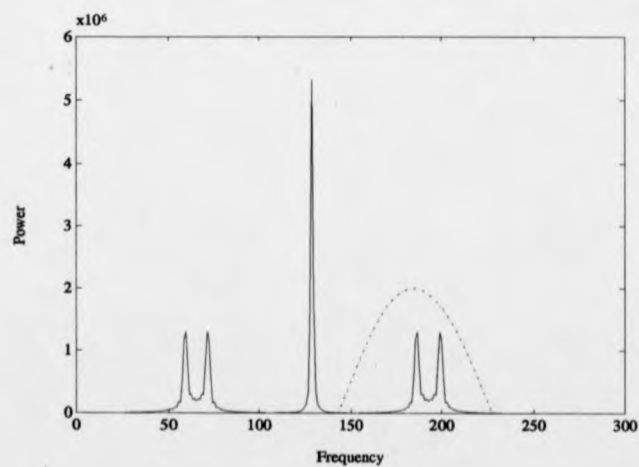


Figure A.9: Power Spectrum of Scan Line and Papoulis Weighting Window

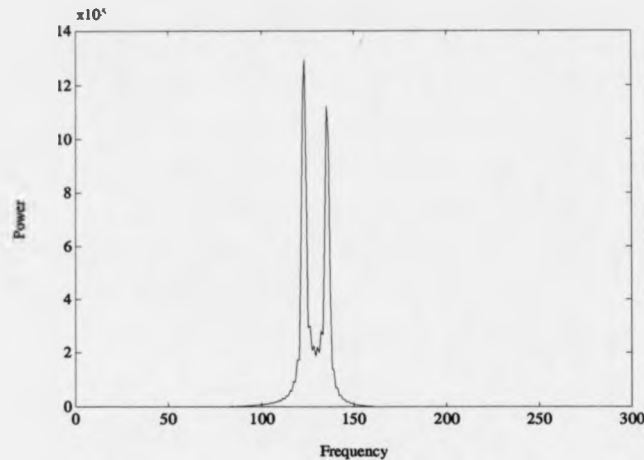


Figure A.10: Translated Side Lobe

This option selects the mode of pixel level phase unwrapping. RAW_DATA indicates that the Minimum Spanning Tree approach should be used. The Minimum Spanning Tree produces a better result than the other options. It is capable of circumventing spike noise without fitting polynomials etc.

FRINGE_COUNT indicates that the fringe scanning or counting method should be used. POLY_CORRECT *n* selects the fringe counting method and attempts to correct some of the errors this method generates by fitting a polynomial of degree *n* to the unwrapped scans. POLY_SMOOTH *n* is similar to POLY_CORRECT *n* except that rather than pushing the values of pixels up or down by 2π to make them fit with the trend of the fitted polynomial (as is the case with POLY_CORRECT *n*), the routine replaces the values of pixels with that computed by the fitted function.

xiv) GRID [*n*] (grid size of mesh in pixels)

This option selects the X grid spacing of the numerical unwrapped output data which is always saved as an ascii file with extension .dat. This file may be used for mesh plots, contour plots etc using

a suitable plotting package. Examples of a mesh plot and contour plot are shown in Figs A.11 and Figs A.12 respectively. The Y grid spacing is computed from the aspect ratio of the image(s) automatically. This means that the data in the output file (.dat) is produced over a square grid, in terms of the real object.

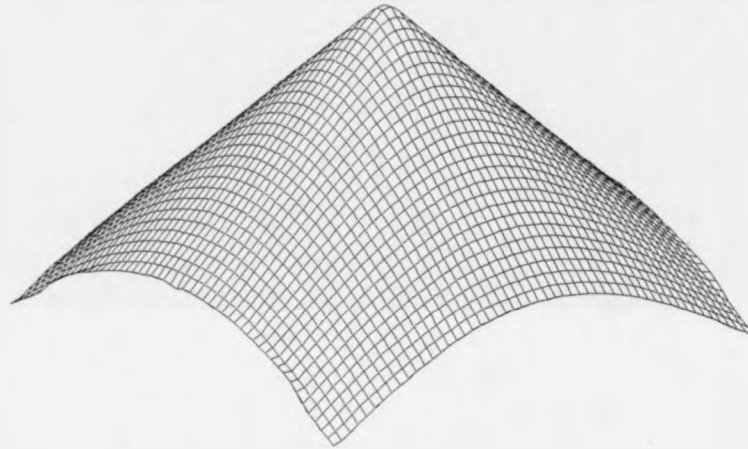


Figure A.11: Example FFT Mesh Plot

xv) OUT_FILES [Output file(s)* prefix]

This option is used to specify the file prefix of all output files.

xvi) USE_RAM or USE_DISC

This option indicates whether to save the intermediate tile solution data on disc as it is computed. This is necessary on the PC to stop the package exhausting the available memory. The Sun version of the code may solve extremely large images using this option.

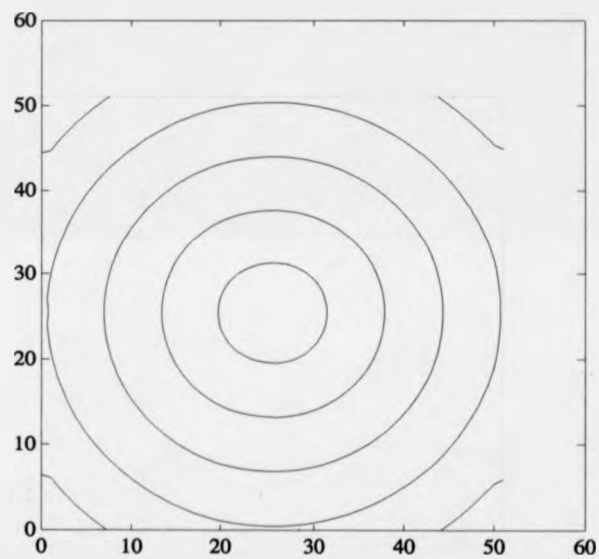


Figure A.12: Example FFT Contour Plot

- xvii) BOUNDS FULLFIELD or [fname] (file containing boundary definition of region to be processed)

This option specifies a boundary for the analysis. A boundary is sometimes useful to speed up the processing by concentrating on the area of interest, or to ensure that erroneous areas of the field are avoided. If analysis of the entire field is required, the option BOUNDS FULLFIELD, should be used. The second form specifies the name of an ascii file containing coordinate data. This file specifies the vertex coordinates of a polygonal boundary within which the analysis code should operate. An example file is shown in Figure A.13. This specifies the full field of a 512 by 512 image. There should be no blank lines in the file.

0
0
511
0
511
511
0
511

Figure A.13: Boundary Example

- xviii) fname (File name of image 1)

This line specifies the file name of the first interferogram. The file should be a TIF image file. This is the only image required for FFT analysis.

- xix) fname (File name of image 2 - not necessary for FFT method)

This line specifies the second image for quasi heterodyne analysis.

- xx) fname (File name of image 3 - not necessary for FFT method)

This line specifies the third image for quasi heterodyne analysis.

A.3.2 An Example Project File

Figure A.14 shows an example project file. This file was used to generate the example images in this document.

```
FFT 0.25
TILE_SIZE 36
BLUR 0
MOD_PERCENT 100.0
NORMALISE_ON
EDGE_DETECTION 1
SAVE_TAN_ON
SAVE_EDGES_ON
SAVE_LOW_MOD_ON
SAVE_GREY_ON
SAVE_TREE_ON
SAVE_FFTS 128,128
SINGLE_TAN
RAW_DATA
GRID 5
OUT_FILES test
USE_DISC
BOUNDS FULLFIELD
test.tif
```

Figure A.14: Example Project File

A.4 Converting To TIF Format

Two utility programs are provided for this purpose. The first utility converts from a raster file to a TIF file. This program is a public domain utility which has been modified by the author of this work. Its usage is shown below

Usage: ras2tif [-vq] <rasterfile> <output tif file>

The second utility 'mktif' is supplied for converting binary image data into the TIF format. The syntax of the utility is shown below.

Usage : mktif [-xn] [-yn] [-an] [-rn] [-w] <input byte image file>
<output tif file>

-xn : This option specifies the size of the image, integer n, pixels in the x direction.

-yn : This option specifies the size of the image, integer n, pixels in the y direction.

-an : This option specifies the aspect ratio, real n, of the image.

-f[file name] : This option allows an image description to be attached to the TIF file. The description is read from the specified file. In order to view the description use the 'tiffin', command.

-rn : This option specifies the output resolution in the x direction. This resolution would be used if the image were printed, for example. It is specified in fractions of an inch. For example, -r72.0, indicates 72 pixels per inch, in the x direction. The y resolution is calculated from the aspect ratio (see above).

-w : This option sets the grey scale so that 0 indicates white and 255 indicates black, rather than the default of 0 meaning black and 255 white.

A.4.1 Defaults for Mktif

If both the x and y pixel resolutions are omitted and the image file length is a square number, $512 * 512 = 262144$, $64 * 64 = 4096$ etc, then the resolution is computed from the square root of the file length.

If one of the resolutions is specified the other is computed by division of the file length e.g. -x64 on a file of length 2048 bytes would give $y = (2048 / 64) = 32$.

If the aspect ratio is not specified it is assumed to be 1.0. An aspect ratio greater than 1.0 indicates that the pixels are smaller in the y direction than in the x. For example, -a2.0 would indicate that if the image were displayed on a corrected video monitor, it would require twice as many pixels in the y direction to equal a similar length in the x direction.

If the -r option is omitted the output resolution is set at 72 pixels per inch. This is the default resolution of postscript laser printers.

If the -w option is omitted the grey scale range will be from (0 = black) to (255 = white).

A.5 Converting From TIF Format

Two utility programs are provided for this purpose. The first utility converts from a TIF file to a sun raster file. This program is a public domain utility which has been modified by the author of this work. Its usage is shown below

Usage: tif2ras -[vq] <input tif file> <rasterfile>

The second utility 'mkbin' allows conversion between the TIF format and a straight 8-bit per pixel binary image.

Usage : mkbin [-xn] [-yn] <input tif file> <output byte image>

-xn : This option specifies the size of the image, integer n, pixels in the x direction. If this is less than the true image width then the output will contain only the leftmost n pixels.

-yn : This option specifies the size of the image, integer n, pixels in the y direction. If this is less than the true image length then the output will contain only the topmost n pixels.

A.6 Creation of FFT Test Image

A utility 'mkfft' is supplied for generating a test image for FFT processing. The test image featured in this document was produced using this program.

Usage : mkfft [-cn] [-fn] [-xn] [-yn] [-an] [-rn] [-w] <output tif file>

The test image is made up of a concentric fringe pattern, modulated by a set of carrier fringes. The spatial frequency of both the underlying concentric fringe pattern and the modulating carrier fringe pattern may be specified. The concentric fringe pattern is centred about the midpoint of the image. The test image was generated with the command;

mkfft -x256 -y256 -f0.25 -c0.025 -r36.0 test.tif

The image without the carrier is shown in Fig A.15. This was generated by the command;

```
mkfft -x256 -y256 -c0.025 -r36.0 conc.tif
```

-cn : This option specifies the concentric fringe frequency, real n, in cycles per pixel.

-fn : This option specifies the carrier fringe frequency, real n, in cycles per pixel.

-xn : This option specifies the size of the image, integer n, pixels in the x direction.

-yn : This option specifies the size of the image, integer n, pixels in the y direction.

-an : This option specifies the aspect ratio, real n, of the image.

-rn : This option specifies the output resolution in the x direction. This resolution would be used if the image were printed, for example. It is specified in fractions of an inch. For example, -r72.0 indicates 72 pixels per inch in the x direction. The y resolution is calculated from the aspect ratio (see above).

-w : This option sets the grey scale so that 0 indicates white and 255 indicates black, rather than the default of 0 meaning black and 255 white.

A.6.1 Defaults for Mkfft

If the carrier frequency option is omitted, then the carrier fringes are not generated. This allows the underlying concentric fringe pattern to be viewed. Likewise omitting the concentric frequency option allows only the carrier fringes to be generated.

If the aspect ratio is not specified it is assumed to be 1.0. An aspect ratio greater than 1.0 indicates that the pixels are smaller in the y direction than in the x. For example, -a2.0 would indicate that if the image were displayed

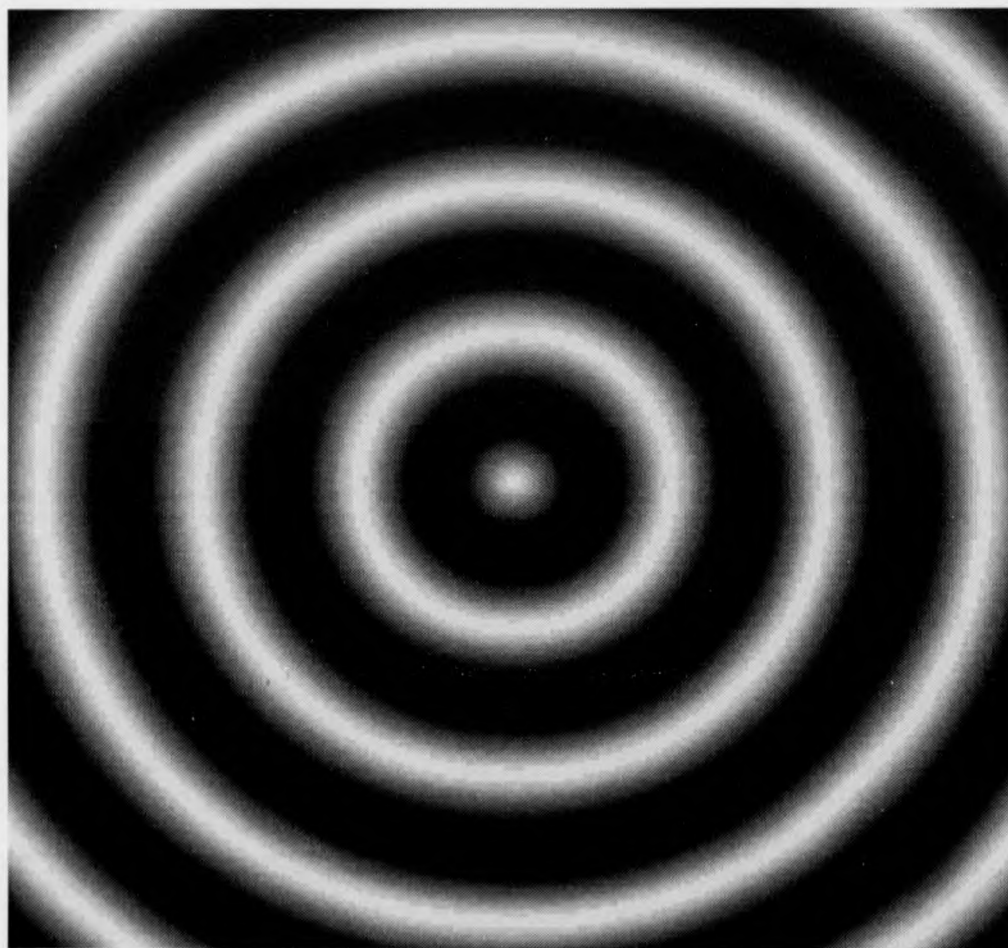


Figure A.15: Example Concentric Fringe Image

on a corrected video monitor, it would require twice as many pixels in the y direction to equal a similar length in the x direction.

If the -r option is omitted the output resolution is set at 72 pixels per inch. This is the default resolution of postscript laser printers.

If the -w option is omitted the grey scale range will be from (0 = black) to (255 = white).

A.7 Displaying Information about a TIF Image

The utility program 'tiffinfo' is provided for displaying information about TIF images. The usage of the program is shown below

Usage: tiffinfo <input tif image file>

A.7.1 Example Use of Tiffinfo

An example of the use of this program is shown below

tiffinfo test.tif

produces the result shown in Figure A.16.

TIFF Directory at offset 0x10008
Image Width: 256 Image Length: 256
Resolution: 36, 36
Bits/Sample: 8
Compression Scheme: none
Photometric Interpretation: "min-is-black"
Host Computer: UNIX SYSTEM
Software: MKFFT : A Test Utility for FRAN
Software Author: Mr. T. R. Judge, Warwick University
Image Description:
FFT Method Test Image, Computer Generated
Carrier Frequency 0.250000
Concentric Frequency 0.025000

Samples/Pixel: 1
Rows/Strip: 31
Planar Configuration: single image plane

Figure A.16: Tifinf Example

Bibliography

- [1] M. Takeda, H. Ina, and S. Kobayashi, "Fourier-transform Method Of Fringe-Pattern Analysis For Computer-based Topography And Interferometry," *Journal Of The Optical Society Of America*, vol. 72, pp. 156-160, 1981.
- [2] R. Thalman and R. Dandliker, "High Resolution Video- Processing For Holographic Interferometry Applied To Contouring And Measuring Deformations," *SPIE ECOOSA*, vol. 492, Amsterdam, 1984.
- [3] D. P. Towers, T. R. Judge, P. J. Bryanston-Cross, "Automatic Interferogram Analysis Techniques Applied To Quasi-heterodyne Holography and ESPI," *Optics and Lasers in Engineering* 14, pp. 239-281, 1991.
- [4] T. R. Judge, PhD Thesis, "Quantitative Digital Image Processing in Fringe Analysis and Particle Image Velocimetry", Engineering Department, Warwick University, 1991.
- [5] G. T. Reid, "Automatic Fringe Pattern Analysis: A Review," *Optics and Lasers in Engineering*, vol. 7, pp. 37-68, 1986/7.

Appendix B

Documentation for the Automated PIDV Image Analysis Package (AP)

B.1 Introduction

The AP package is designed to allow the semi-automated analysis of images from Particle Image Displacement Velocimetry (PIDV). The package is implemented for IBM PCs and compatible machines.

Velocity is coded by the displacement between double exposure particle images.

The package operates over Tagged Image Format Files (TIFs) with either 8 or 1 bit per pixel of intensity resolution. Particle images are expected to be white on a dark background (in the case of 8 bit TIFs) or in the case of 1 bit TIFs binary 1 on a background of zero. Such files would normally be generated by flat bed scanners, for example. The most convenient format for PIDV data is the 1 bit per pixel compressed form. This is known as Macintosh Packbits encoding. In this case image files are very small, typically around 80K for a 10 inch by 8 inch print scanned at 300 pixels per inch.

B.2 User Interface

The package is menu driven. Help for any option is available by placing the selector bar over an option and pressing the F1 key, Figure B.1.

File Tif	File Piv	View Tif	View Vec	Zoom In	Filter	Process	Batch
				PIV Parameters			
				Scale (1cm in scanned image == ? cm in PIV experiment)			
				Threshold intensity level for 8 bit TIF			
				mInimum particle size (square pixels (TIF))			
				mAXimum particle size (square pixels (TIF))			
				minimum Feature size (square pixels (TIF))			
				Pulse separation (micro seconds)			
				miNimum velocity accepted (metres/second)			
				maXimum velocity accepted (metres/second)			
				miniMum angle accepted (0 = north, 90 = east etc)			
				maximUm angle accepted (0 = north, 90 = east etc)			
				set Box size in microns for direction validation			
				set vector plOt scaling factor			
F1-Help	F2-Save Particles	F3-Open PIV Files	F4-Dos Command	F5-Core Left			

Figure B.1: PC Screen Display for AP Package

B.3 Main Menu

The main menu is found along the top line of the screen, see Figure B.1. The various options are accessed by moving the selector bar with the cursor keys, left and right. Each of the options is described below.

B.3.1 File Tif

Selection of this option accesses a menu which is specifically concerned with the input TIF file. The first option allows a new TIF file to be opened which automatically closes down the image then being worked upon.

In addition to opening the TIF file, the disc is interrogated for any other

files associated with the image. If the image had, for example, been analysed previously then the analysed particle data would be loaded for viewing.

This option is also available from function key F3.

The File Tif menu also provides an option to view details about the TIF file currently being operated upon, for example its x and y pixel dimensions, the resolution at which it was digitised, the number of bits per pixel etc.

B.3.2 File Piv

Selection of this option accesses a menu which is concerned with the various files created during PIDV image processing. After processing via the Process option (described later), the output consists of a number of files. These include particle data files, plot files, feature data files and text files describing the image.

It is sometimes desirable to create one or more of these files without having to reprocess the entire image. The PIDV data held in the PC's memory is stored in terms of particle positions. It is only when a file is output that velocities are computed. The then current values of pulse separation, scaling etc are employed. So, for example, to explore the effect on the results of a slight variance in pulse separation, the pulse separation would simply be changed via the relevant menu option and a request made to save the appropriate file (which would then be in terms of the new pulse separation).

B.3.3 View Tif

This option permits the input TIF file to be displayed on the PC screen. The TIF image is squashed so that the whole file is fitted on the screen. The aspect ratio is retained.

If a closer inspection of the TIF image is required then the Zoom In option may be used (see below).

If selection of the View Tif option is made after the image has been processed then the velocity vectors are drawn in over the TIF image.

B.3.4 View Vec

This option is similar to the one above, except that it permits the velocity vectors to be viewed in isolation from the original TIF image. This can give a clearer display.

B.3.5 Zoom In

This option permits the TIF image and processing results to be displayed at high resolution. In this case every pixel of the digitised image is mapped to a pixel on the PC screen, normally this means that one pixel on the screen is equivalent to $\frac{1}{300}$ of an inch in the digitised print.

If the image is larger than the PC screen (as is normally the case) then the cursor keys permit the window of the PC screen to be moved around the image.

This option is only available after the TIF image has been processed. Initially only the velocity vectors themselves are displayed, with the corresponding speed in metres per second. That is the image from which they result is not displayed. This permits rapid movement around the image with the cursor keys. If the underlying image is required then this may be invoked as an underlay to the velocity vectors by pressing the 'F' key (for feature). This is a toggle. If the PC window is moved across the image, then the TIF file will be displayed together with the velocity vectors until the toggle is switched off by another stroke on the 'F' key.

In order to display the angle of the velocity vectors, instead of the velocity, the toggle key 'T' may be used.

B.3.6 Filter

In order to operate the package a number of factors must be supplied by the user. These are listed below, and shown on the pull down menu in Figure B.1. This menu is accessed by selecting the 'Filter' option from the list at the top of the screen.

B.3.6.1 Image Scaling

This option is used to specify the scale relationship of the input image to that of the real experiment.

The input image would normally be a photographic print of the PIDV negative, digitised by the scanner. The TIF image format is a widely accepted standard for the output from such devices. It contains as part of its specification the resolution of the digitisation process used to create the image. This is typically of the order of 300 pixels per inch. From this piece of information and the x and y pixel dimensions of the image, the size of the digitised print may be calculated. That is the dimensions of the photographic print are recorded automatically.

The scaling factor is required to specify the scale relationship between the photographic print and the real spatial dimensions of the PIDV experiment. Some record of the scale of the experiment must be made at the time of the experiment. One method is to record a test shot of a centimetre rule placed in the plane of the light sheet. The scaling factor is determined by measurement of the apparent size of a one centimetre length of the ruler in the test shot, printed at the same scale as the PIDV negative. The factor represents the reduction ratio from print to spatial dimensions. As an example, if 1cm in the real experiment occupied 10cm on the photographic print then the scaling factor should be specified as 0.1.

B.3.6.2 Threshold Intensity Level for 8 Bit TIF File

The first processing step for grey scale (as opposed to bilevel) PIDV images is thresholding. Particles have a much higher intensity than the background level due to the light reflected from them, thresholding is employed to spatially delimit the particle images from the background. The thresholding process converts the grey scale image to a bilevel representation in which the high intensity pixels belonging to particles are denoted by a pixel value of 1 and the background as a pixel value of 0.

The filter menu option 'Threshold intensity level for 8 bit TIF' permits the threshold level to be specified as a percentage of full scale. The effect of a specific threshold setting on the detection of particle images may be viewed

by selecting the 'View Tif' option. The option displays an 8 Bit TIF file as a bilevel thresholded image with the then current threshold setting.

B.3.6.3 Particle Size

Particles are sized during a flood fill of illuminated pixels. This means that the pixel area of illuminated areas of the image is known.

In PIDV images it is possible that areas of glare or reflection will be present. An automatic process of avoiding such features is necessary. This is implemented by fixing a limit on the size of objects that are to be considered particles. Any object in the image which exceeds this size is ignored. A similar lower limit may also be defined.

B.3.6.4 Feature Size

Features are defined as large areas of glare or reflection. The positions of such features can be a valuable aid in registration, that is, in recognising the portions of the flow to which the PIDV data corresponds. During the sizing procedure mentioned above, the pixels describing the outline of the filled areas are recorded. If upon completion of the filling process it is found that the area sized is greater than the specified minimum feature size, then the coordinates of the outline of the feature are recorded. These recorded outline coordinates are subsequently added to velocity vector plots etc, as a reminder of the area considered.

B.3.6.5 Pulse Separation

This option is used to specify the laser pulse separation in time between recording of the first and second particle images which go to make a particle pair. The separation should be specified in microseconds.

B.3.6.6 Minimum / Maximum Velocity Accepted

This option is used to specify the band limits of velocity in the PIDV image. This is necessary to permit a partial resolution of particle pair ambiguities, see Figures B.2, B.3 and B.4.

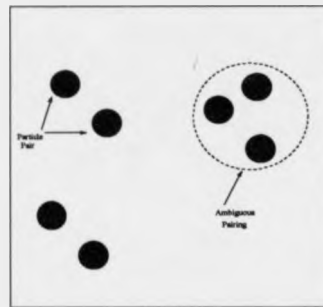


Figure B.2: Schematic of Particle Pairs Showing an Ambiguous Pairing Problem

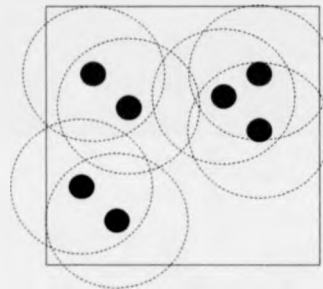


Figure B.3: Schematic of Particle Pairs, Circles Define the Maximum Distance that a Particle could have Moved, Pulse to Pulse

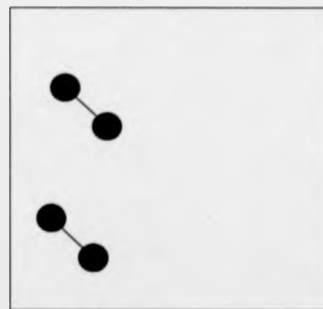


Figure B.4: Schematic of Particle Pairs, Ambiguous Pairings are Deleted and Velocities Computed for the Remaining Pairs

B.3.6.7 Minimum / Maximum Angle Accepted

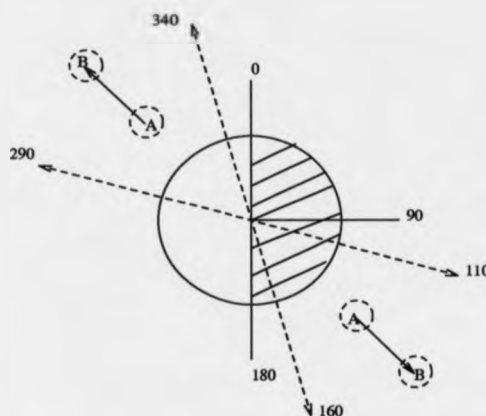


Figure B.5: Specifying Band Limits for Direction

In a similar fashion to the velocity band limits these options permit a directional range to be specified to aid in the resolution of particle pair ambiguities.

Consider Figure B.5. A and B mark the position of a particle at the time of the first and second laser pulse, respectively. The coding scheme for the particles is not sufficient to specify which particle image corresponds to the first pulse and which to the second. There is therefore a 180 degree ambiguity in the direction in which the particle was moving.

To specify the band limits for flow direction it is sufficient to specify the minimum and max angle in the range 0 to 180 degrees, the shaded area of Figure B.5. For example specifying a range of between 110 and 160 degrees will automatically included pairs which appear to be moving on a bearing of between $110+180 = 290$ to $160+180 = 340$ degrees.

B.3.7 Process

This option initiates processing of the then current TIF image.

During the first stage of processing the package looks for particle images and records their position. This data is retained on disc. Suppose the PIDV image requires reprocessing to better tune the analysis. If parameters un-

connected with this initial stage of processing are modified then this step is not repeated. For example, suppose the band limits for velocity are adjusted. This has no effect on the store of candidate particles considered for pairing, so the detection stage of processing is bypassed. In practice this means that the effect of different band limits on the analysis may be evaluated very quickly.

Pairing of the particles is performed during the second stage of processing. This phase is also responsible for the detection of pairing ambiguities.

Files with the extensions given below are involved in the analysis process. The file prefix for these files is extracted from the source TIF image file. For example, if the source image is named 'image.tif', then the prefix would be 'image'. 'tif' is the standard extension for TIF files.

- i) cfg : This file contains scaling and other information concerning the image under analysis. If this file exists when a TIF image is loaded then the scaling information it contains is automatically recalled. This file is created whenever the image is reprocessed.
- ii) dat : This file is produced in order that the velocity vectors may be plotted from a suitable package. The file format was designed for use with 'matlab'. The file is in ASCII format. Each line contains the following information

X coordinate in microns of particle pair mid point.

Y coordinate in microns of particle pair mid point.

Velocity of particle pair in metres per second.

Velocity vector angle in degrees (0 = north).

The final line of the file is special. It contains the pulse separation of the laser in microseconds and the scaling factor for the arrows of the plot, followed by zero, zero. Therefore the last line also contains four numbers.

- iii) par : This is a binary file. It contains a list of C structures, one for each particle. This structure is shown below

```
typedef struct xy_particle {
```

```

int xp,yp;
long mass;
char delete;
struct xy_particle huge* pred;
struct xy_particle huge* succ;
} XY_PARTICLE;

```

The particle data in this file is unpaired. The file represents the store of candidate particles. That is, unless the limits upon particle size are changed from within the Filter Menu, any PIDV solutions are computed from the data in this file. If the limits for particle size are changed then this file is automatically regenerated from the TIF file.

- iv) prs : This file contains data in the same format as the 'par' file except that each consecutive pair of structures represents a particle pairing. This file represents the current field solution. It is regenerated from the 'par' file, taking account of the Filter options, whenever the Process option is selected.
- v) crd : This is an ASCII file. It contains the integer X, Y TIF coordinates of the centres of all particles in the 'par' file, multiplied by 2.
- vi) daf : This is an ASCII file, it contains the X, Y coordinates in microns of all points found on the edges of features. It is used in conjunction with the 'dat' file to produce velocity vector plots.
- vii) fet : This is a binary file containing the integer TIF coordinates of points on the edges of features.
- viii) vec : This is an ASCII file. This is the main textual description of the result of PIDV analysis. It contains most of the information extracted during PIDV processing in the form of a table and statistics such as mean velocity, mean vector angle, mean particle size, plus standard deviations.

B.3.8 Batch

This option permits batch processing of a number of PIDV images.

In order to use this option a text file must first be created containing the names of the TIF images to be processed, including the extension .tif. One file name should be specified per line. Batch processing is started by selecting the Batch option and entering the file name of this text file.

Bibliography

- [1] R. J. Adrian, "Particle Image Displacement Velocimetry", Lecture Series 1988-06, von Karman Institute for Fluid Dynamics, Chaussee de Waterloo 72, B - 1640 Rhode Saint Genese, Belgium, March 21-25, 1988.
- [2] T. R. Judge, PhD Thesis, "Quantitative Digital Image Processing in Fringe Analysis and Particle Image Velocimetry", Engineering Department, Warwick University, 1991.

**Quantitative Digital Image Processing in
Fringe Analysis and Particle Image
Velocimetry (PIV)**

Volume II (Program Listings)

Thomas Richard Judge
Department of Engineering,
University of Warwick

This thesis is submitted for the degree of
Doctor of Philosophy

December 15, 1991

Declaration

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been done by myself except where stated otherwise.

Thomas Richard Judge

To Mum, Dad, Ruth and Louise

Chapter 1

Software Listing for FRANSYS Main Program

1.1 Introduction

This chapter gives a software listing for the main program in the FRANSYS analysis package.

The main program is divided into a series of modules. These modules are listed below.

- i) `franheader.h` : This header file contains some definitions which are used throughout the program.
- ii) `computewrapped.c` `computewrapped.h` : This module is responsible for computing the wrapped phase maps (by both the FFT and Phase Stepping Methods).
- iii) `correctoffsets.c` `correctoffsets.h` : This module handles correction of tile offsets. That is, it occasionally happens that the offset given by the MST for a bad tile can be improved. This is a post processing operation which considers the number of tiles, neighbouring the bad tile, which suggest the same tile offset.
- iv) `dynamicarray.c` `dynamicarray.h` : This module handles the dynamic memory allocation of variably sized two dimensional arrays.

- v) `fringe`.c `fringe`.h : This module implements a fringe counting algorithm. The algorithm operates on a per tile basis (See FRANSYS documentation).
- vi) `graph`.c `graph`.h : This module implements the graph data structure as a linked list, with double links.
- vii) `imageprepro`.c `imageprepro`.h : This module is responsible for image preprocessing, that is normalisation, averaging and median filters.
- viii) `main`.c `main`.h : This is the main control module. It also contains the adaptive Sobel edge detector.
- ix) `polysmooth`.c `polysmooth`.h : This module implements polynomial smoothing.
- x) `tileelements`.c `tileelements`.h : This module controls the assignment of tiles to the whole field array.
- xi) `unwrap`.c `unwrap`.h : This module controls the phase unwrapping process at both the tile and pixel levels. It computes the weighting factors and minimum spanning trees.
- xii) Makefile: This file may be used to compile the program on a Unix system. The program may be compiled for an IBM PC or Compatible using Turbo C Version 2.0 or higher. The only change to the program then required is that the system definition in the file '`franheader.h`' should be changed from '`#define unixc`' to '`#define turboc`'.

The program is dependent upon the availability of the TIFF Library for SPEC 5.0, Release 2.1 written by Sam Leffler. There is a mailing list associated with this library `tiff@ucbvax.berkeley.edu`. This may be joined by sending a message to `tiff-request@ucbvax.berkeley.edu`. Sam Leffler may be contacted directly at `sam@ucbvax.berkeley.edu`.

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5
6  #define TILE_ABOVE (0)
7  #define TILE_BELOW (1)
8  #define TILE_RIGHT (2)
9  #define TILE_LEFT (3)
10 #define PIXEL_ABOVE (0)
11 #define PIXEL_BELOW (1)
12 #define PIXEL_RIGHT (2)
13 #define PIXEL_LEFT (3)
14 #define NO_NEIGHBOUR (-1)
15 #define TILE_OFFSET_UNDEFINED (0)
16 #define TILE_OFFSET_COMPUTED (1)
17 #define MAXVERT 50
18 #define MAX GREY_LEVEL ((unsigned char) 255)
19 #define MIN GREY_LEVEL ((unsigned char) 0)
20 #define NIMAGES 3
21 #define PIF ((float) (3.141592654))
22 #define DELTA FOR_EDGE (PIF)
23 #define NO_NEIGHBOURS LEFT 4
24 #define ABOVE HIGH_THRESHOLD 8
25 #define STEP_UP_EDGE_LEFT TO RIGHT 16
26 #define STEP_DOWN_EDGE_LEFT TO RIGHT 32
27 #define STEP_UP_EDGE_TOP TO BOTTOM 64
28 #define STEP_DOWN_EDGE_TOP TO BOTTOM 128
29 #define LOW_MOD ((float) (4.0*PIF))
30 #define LIMIT HEIGHT ((float) (1.0e+36))
31 #define DEAD_PIXEL ((float) (1.0e+37))
32 #define THRESHOLD (51)
33 /* THRESHOLD = 20 percent of 255 */
34 #define RUN COMPLETED ((int) 0xAAA)
35 #define GREYSCALE_WHITE ((unsigned char) 255)
36 #define GREYSCALE_GREY ((unsigned char) 192)
37 #define GREYSCALE_BLACK ((unsigned char) 0)
38 #define GREYSCALE_TREEMARK ((unsigned char) 64)
39 #define GREYSCALE_LOWEDGE ((unsigned char) 32)
40 #define GREYSCALE_HIGHEdge ((unsigned char) 128)
41 #define TOUCHES_TOP (0)
42 #define TOUCHES_BOTTOM (1)
43 #define TOUCHES_LEFT (2)
44 #define TOUCHES_RIGHT (3)
45
46 /* #define turboc */
47
48 #define uninc
49
50 /* #define debug */
51
52 extern void Exit();
53
54 #ifdef uninc
55 extern char* calloc();
56 #define MAXPATH 128
57 #define SEEK_END 2
58 #define prn printf
59 #define printat(y,x) move(y,x)
60 #define clrscr clear
61 #endif
62
63 #ifdef turboc
64 #define prn printf
65 #define printat(y,x) gotoxy(x,y)
66 #endif

```

```

67 typedef unsigned char** IMAGE;
68
69 typedef float** HEIGHTS;
70
71 typedef float** FLOATS;
72
73 typedef float** PHASE;
74
75 typedef unsigned char** BYTES;
76
77 typedef float** OVERLAP;
78
79 typedef struct neighbour_data
80 { float diffs[4]; int ndiff[4]; }
81 NEIGHBOUR_DATA;
82
83 typedef struct tile_elem {
84     int tx,ty;
85     HEIGHTS tile;
86     long scan_dir;
87     unsigned int edge_end_count;
88     unsigned int dead_pixel_count;
89     float min_height;
90     float max_height;
91     int status;
92     float offset;
93     NEIGHBOUR_DATA n;
94 } TILE_ELEM;
95
96 typedef struct analysis_parameters {
97     char files_prefix[80];
98     char file_names[3][80];
99     char config_name[80];
100     char bounds_name[80];
101     float phase_freq;
102     float carrier_freq;
103     float ASPECT;
104     float CRIT_THRES_VAL;
105     float gridx,gridy;
106     float threshold;
107     int blur;
108     int edge_detect;
109     int median;
110     unsigned carrier_raster;
111     int is_fringe_doubling;
112     int is_phase_stepping;
113     int is_ready_wrapped;
114     int normalise;
115     int nvertex;
116     int save_edge_detect;
117     int save_grey;
118     int save_mod;
119     int save_phase;
120     int save_tree;
121     int save_end;
122     int smooth;
123     int unwrap_tile_by_mst;
124     int use_ram;
125     int xv[MAXVERT];
126     int yv[MAXVERT];

```

```

133 unsigned max_dead_pixel_count;
134 unsigned NFILES;
135 unsigned NFILESf;
136 unsigned order_of_polyfit;
137 unsigned PIXM;
138 unsigned PIXY;
139 unsigned TILESIZEx;
140 unsigned TILESIZY;
141 unsigned XSTEP;
142 unsigned YSTEP;
143 File* tile_solutions;
144 File* tile_array_dat;
145 } ANALYSIS_PARAMETERS;
146
147 extern ANALYSIS_PARAMETERS ap;

```

Fringe Analysis	./fran/memory.h	Page 1
<pre> 1 /* 2 * Copyright (c) 1991 by Tom Judge. 3 * All rights reserved. 4 */ 5 #ifdef unixc 6 7 extern char* valloc(); 8 9 #define Walloc(size) malloc(size) 10 11 #define Calloc(nelem, elsize) calloc((nelem), (elsize)) 12 13 #define Free(p) free(p) 14 15 #endif 16 17 18 #ifdef turboc 19 20 #define Walloc(size) malloc(size) 21 22 #define Calloc(nelem, elsize) calloc((nelem), (elsize)) 23 24 #define Free(p) free(p) 25 26 #endif 27 </pre>		

```
1  /*  
2  * Copyright (c) 1991 by Tom Judge.  
3  * All rights reserved.  
4  *  
5  */  
6  extern void Initialise_Phase_Step_Tan_Fringe_Computation();  
7  extern void Compute_Phase_Step_Row_Of_Tan_Fringes();  
8  extern void Close_Phase_Step_Tan_Fringe_Computation();  
9  extern void Initialise_Fft_Tan_Fringe_Computation();  
10  extern void Compute_Fft_Row_Of_Tan_Fringes();  
11  extern void Close_Fft_Tan_Fringe_Computation();  
12  extern void Setup_Mat_Fft_and_Fringe_Computation();  
13  extern void Perform_One_Dimensional_Fft_On_Scan_Line();  
14  extern void Initialise_Wrapped_Computation();  
15  extern void Compute_Wrapped_Row_Of_Tan_Fringes();  
16  extern void Close_Wrapped_Computation();  
17  
18  extern float carrier_freq;  
19  extern int cutoff_print_count;  
20  extern char prefix[];  
21  extern int* threshold_results;  
22  extern void Read_Scanline();  
23  extern Tiff* Create_Edge_TIFF();  
24  extern char Get_Time();  
25  extern void Set_Scanline();  
26  extern void Setup_TIFF_Header();
```



```

1  * Copyright 1991 by Tom Judge.
2  * All rights reserved.
3
4  */
5
6  #include <stdio.h>
7  #include <ctype.h>
8  #include <math.h>
9  #include <string.h>
10 #include <time.h>
11 #include <assert.h>
12 #include <franhdr.h>
13 #ifdef turbo
14 #include <conio.h>
15 #include <stdlib.h>
16 #include <dos.h>
17 #endif
18 #include <alloc.h>
19 #ifdef unix
20 #include <malloc.h>
21 #include <cursor.h>
22 #endif
23 #include "tiffio.h"
24 #include "memory.h"
25 #include "correctcoeffs.h"
26 #include "fringeout.h"
27 #include "dynamarray.h"
28 #include "polyasynth.h"
29 #include "polyasynth.h"
30 #include "tiffio.h"
31 #include "computewrapped.h"
32 #include "main.h"
33
34 #define TWO_PI (5.28318530717959)
35
36 int
37 int i;
38 TIFF* in[3]; raster[3];
39 unsigned char* raster[3];
40 unsigned short* raster[3];
41 TIFF* out_top;
42 TIFF* out_bot;
43
44 int
45 ap, yp;
46 double i0, i1theta, i2theta;
47 char name[10];
48 float x0, offset;
49 double top_bottom;
50 double critical_size;
51 double cos_theta_minus_i;
52 double cos_2theta_minus_i;
53 double sin_theta;
54 double sin_2theta;
55 double i2theta_minus_i0;
56 double i2theta_minus_i0;
57 int n_scan_lines;
58 int first_time;
59 unsigned n_scans_read;
60 int dump_ifits_begin_scan;
61 int dump_ifits_end_scan;
62 int threshold_results; /* Used for detecting absence of carrier */
63 unsigned n_padded_zeros;
64 unsigned n_padded_zeros;
65 char data_time[80];
66 char document[256];

```

```

67 int tile_rasters_completed;
68 FILE* wrapped_phase_map;
69 float* wrapped_raster;
70
71 #ifdef turbo
72 struct date* datep;
73 struct time* timep;
74 #endif
75
76 /* variables for FFT tan fringe computation */
77 float* window_spatial;
78 float* window_fourier;
79
80 /*
81 Possible Sigma Coefficients for Windows
82
83 Do-nothing = { 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 }
84 Exact Papoulis = { 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0 }
85 Exact Hanning = { 0.5, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0 }
86 Hamming = { 0.539, 0.0, 0.461, 0.0, 0.0, 0.0, 0.0 }
87
88 */
89
90 float sigma_coeffs_spatial[] = { 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
91 int n_coeffs_spatial = 7;
92 float sigma_coeffs_fourier[] = { 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
93 int n_coeffs_fourier = 7;
94
95 float* scan;
96 char prefix[255];
97 float* powers;
98
99 short width;
100 height;
101 long rowsperstrip;
102 short photometric;
103 short samplesperpixel;
104 long bps;
105 long xres;
106 double yres;
107 double depth;
108 short
109 #ifdef turbo
110
111 void Set_Date_Time()
112 {
113 time_t;
114 struct time d_time;
115 struct date d_date;
116 struct tm* local;
117
118 getdate(&d_date);
119 gettime(&d_time);
120
121 t = dostounix(&d_date, &d_time);
122 local = localtime(&t);
123 sprintf(date_time, "%s", asctime(local));
124 }
125
126 #endif
127
128 void Setup TIFF Header( tif_out, tif_in, additional_description,
129 bipsersample )
130 TIFF* tif_in;
131 TIFF* tif_out;
132 char* additional_description;

```

Fringe Analysis	./fran/computewrapped.c	Page 3
133	short bitspersample;	
134	{	
135	char* new_description;	
136	char* original_description;	
137		
138	width = tif->tif_dir.td_imagewidth;	
139	height = tif->tif_dir.td_imagelength;	
140	original_description = tif->tif_dir.td_imagedescription;	
141		
142	samplesperpixel = 1;	
143		
144	depth = samplesperpixel * 8;	
145		
146	photometric = PHOTOMETRIC_MINISBLACK;	
147		
148	bpsl = ((long) width * (long) bitspersample * (long) samplesperpixel) /	
149	8L;	
150		
151	rowsperstrip = (8096L / bpsl);	
152		
153	xres = (double) tif->tif_dir.td_xresolution;	
154	yres = (double) tif->tif_dir.td_yresolution;	
155		
156	TIFFSetField(tif_out, TIFFTAG_IMAGEWIDTH, width);	
157	TIFFSetField(tif_out, TIFFTAG_IMAGELENGTH, height);	
158	TIFFSetField(tif_out, TIFFTAG_BITSPERSAMPLE, bitspersample);	
159	TIFFSetField(tif_out, TIFFTAG_COMPRESSION, tif->tif_dir.td_compression);	
160	TIFFSetField(tif_out, TIFFTAG_PHOTOMETRIC, photometric);	
161		
162	sprintf(document, "Project File : %s", ap.config_fname);	
163		
164	TIFFSetField(tif_out, TIFFTAG_DOCUMENTNAME, document);	
165		
166	#ifdef turbo	
167	Set Date Time();	
168	TIFFSetField(tif_out, TIFFTAG_DATETIME, date_time);	
169	TIFFSetField(tif_out, TIFFTAG_HOSTCOMPUTER, "IBM PC or Compatible");	
170	#endif	
171		
172	#ifdef unix	
173	TIFFSetField(tif_out, TIFFTAG_HOSTCOMPUTER, "UNIX SYSTEM");	
174	#endif	
175		
176	TIFFSetField(tif_out, TIFFTAG_SOFTWARE,	
177	"FRAN: A Fringe Analysis Package, By T. R. Judge, Warwick University (1991)."	
178);	
179		
180	TIFFSetField(tif_out, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);	
181		
182	if (original_description != NULL)	
183	{	
184	new_description = (char*)	
185	Malloc(strlen(additional_description) + strlen(original_description));	
186		
187	if (new_description == NULL)	
188	frn_Error(fran_module, "Couldn't allocate memory for new image descriptio	
189	n");	
190	strcpy(new_description, additional_description);	
191	strcpy({ new_description + strlen(additional_description) },	
192	original_description);	
193		
194	TIFFSetField(tif_out, TIFFTAG_IMAGEDESCRIPTION, new_description);	
195		
196	}	

Fringe Analysis	./fran/computewrapped.c	Page 4
197	else	
198	TIFFSetField(tif_out, TIFFTAG_IMAGEDESCRIPTION, additional_description);	
199		
200	TIFFSetField(tif_out, TIFFTAG_SAMPLESPERPIXEL, samplesperpixel);	
201		
202	if (rowsperstrip > 0L)	
203	TIFFSetField(tif_out, TIFFTAG_ROWSPERSTRIP, rowsperstrip);	
204	else	
205	TIFFSetField(tif_out,	
206	TIFFTAG_ROWSPERSTRIP, ((unsigned long) 0xfffffff);	
207		
208	TIFFSetField(tif_out, TIFFTAG_XRESOLUTION, xres);	
209	TIFFSetField(tif_out, TIFFTAG_YRESOLUTION, yres);	
210	TIFFSetField(tif_out, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);	
211		
212	void Read Scanline(tif, buffer, y)	
213	TIFF* tif;	
214	unsigned char* buffer;	
215	unsigned y;	
216	{	
217	unsigned i;	
218		
219	if (tif->tif_dir.td_bitspersample == 8)	
220	{	
221	TIFFReadScanline(tif, buffer, (u_int) (y), (u_int) 0);	
222	if (tif->tif_dir.td_photometric == PHOTOMETRIC_MINISWHITE)	
223	for (i = 0; i < tif->tif_dir.td_imagewidth; i++) buffer[i] =	
224	igned char((255-buffer[i]);	
225	}	
226		
227	if (tif->tif_dir.td_bitspersample == 16)	
228	{	
229	unsigned short* short_buffer;	
230		
231	short_buffer = (unsigned short*) buffer;	
232		
233	TIFFReadScanline(tif, short_buffer, (u_int) (y), (u_int) 0);	
234		
235	if (tif->tif_dir.td_photometric == PHOTOMETRIC_MINISWHITE)	
236	for (i = 0; i < tif->tif_dir.td_imagewidth; i++)	
237	short_buffer[i] = (unsigned short) (65535 - short_buffer[i]);	
238	}	
239		
240	TIFF* Create_Edge_TIFF()	
241	{	
242	TIFF* out_edg;	
243		
244	sprintf(fname, "%s.edg", ap.files_prefix);	
245		
246	out_edg = TIFFOpen(fname, "w");	
247		
248	if (out_edg == NULL) frn_Error(fran_module, "Couldn't open edge detection f	
249	ile");	
250		
251	if (ap.save_tan == 1)	
252	setup TIFF Header(out_edg, out_tan,	
253	Edge Detection of Wrapped Phase Map\n", 8);	
254	else	
255	Setup TIFF Header(out_edg, in[0],	
256	"Edge Detection of Wrapped Phase Map\n\n", 8);	
257		
258	return(out_edg);	
259		
260	}	

```

261 void Initialise_Phase_Step_Tan_Fringe_Computation
262 {theta,save_tan,save_mod,pref}
263 double theta;
264 int save_tan;
265 int save_mod;
266 char* pref;
267 {
268     char string[256];
269     unsigned temp;
270
271     strcpy( prefix, pref );
272
273     cos_theta_minus_1 = cos( theta ) - 1.0;
274     cos_2theta_minus_1 = cos( (2.0*theta) ) - 1.0;
275     sin_theta = sin( theta );
276     sin_2theta = sin( (2.0*theta) );
277
278     scale_tan = 65535.0/(2.0*PIF);
279     tan_offset = PIF;
280
281     critical_size = 10.0;
282
283     ap.PIXX = 0;
284     ap.PIXY = 0;
285
286     for( i = 0; i < 3; i++ )
287     {
288         in[i] = TIFFOpen( (char*) ap.file_names[i], "r" );
289         if ( in[i] == NULL )
290             continue;
291
292         sprintf( string, "Couldn't open image file: %s", ap.file_names[i] );
293         Fran_Error( fran_module,string );
294     }
295
296     /* TIFFPrintDirectory( in[i], stdout, 0, 0, 0 ); */
297
298     temp = (unsigned) in[i]->tif_dir.td_imagelength;
299
300     if ( temp > ap.PIXX ) ap.PIXX = temp;
301
302     temp = (unsigned) in[i]->tif_dir.td_imagewidth;
303
304     if ( temp > ap.PIXX ) ap.PIXX = temp;
305
306     raster[i] = (unsigned char*)
307         Malloc( TIFFScanlinesize( in[i] ) );
308
309     if ( raster[i] == NULL )
310         Fran_Error( fran_module, "Ran out of memory trying to allocate
311         tif raster" );
312
313     ap.NTILESX = (ap.PIXX/ap.XSTEP);
314     if ( ( ap.PIXX % ap.XSTEP ) != 0 ) ap.NTILESX++;
315
316     ap.NTILESY = (ap.PIXY/ap.YSTEP);
317     if ( ( ap.PIXY % ap.YSTEP ) != 0 ) ap.NTILESY++;
318
319     n_scan_lines = (int) (ap.TILESIZE + 2);
320
321     phase = Create_Phase_Array( ap.PIXX, (unsigned) n_scan_lines );
322
323     first_time = 1; n_scans_read = 0;
324
325     ap.ASPECT = (float) ( in[0]->tif_dir.td_yresolution /

```

```

326     in[0]->tif_dir.td_xresolution );
327
328     printat( 24, 25 );
329
330     prn( "Image Size : X=%u, Y=%u Pixels\n", ap.PIXX, ap.PIXY );
331
332     if ( save_tan == 1 )
333     {
334         sprintf( frame, "%s.tan", pref );
335         out_tan = TIFFOpen( frame, "w" );
336
337         if ( out_tan == NULL ) Fran_Error( fran_module, "Couldn't open wrapped ph
338         ase map file" );
339
340         Setup TIFF Header( out_tan, in[0],
341             "\nMirrored Phase Map Generated by Phase Stepping\n", 16 );
342
343         raster_tan = (unsigned short*) Malloc( TIFFScanlinesize( out_tan ) );
344
345         if ( raster_tan == NULL )
346             Fran_Error( fran_module, "Ran out of memory trying to allocate tif ra
347             ster" );
348
349         if ( save_mod == 1 )
350         {
351             raster_mod = (unsigned char*) Malloc( TIFFScanlinesize( in[0] ) );
352
353             if ( raster_mod == NULL ) Fran_Error( fran_module, "Ran out of memory tr
354             ying to allocate tif raster" );
355
356             sprintf( frame, "%s.mod", pref );
357             out_mod = TIFFOpen( frame, "w" );
358
359             if ( out_mod == NULL ) Fran_Error( fran_module, "Couldn't open low modula
360             tion file" );
361
362             Setup TIFF Header( out_mod, in[0],
363                 "\nLow Modulation Pointa\n", 8 );
364
365             void Compute_Wrapped_Row_Of_Tan_Fringes(save_tan
366             int save_tan;
367             int y_start;
368             int i;
369
370             if ( first_time != 1 )
371             {
372                 for( i = 0; i < 6; i++ )
373                 {
374                     j = (ap.TILESIZE+2) - 6 + i;
375                     for( xp = 0; xp < ap.PIXX; xp++ )
376                         phase[xp][i] = phase[xp][j];
377                 }
378                 y_start = 6;
379             }
380             else
381             {
382                 y_start = 6;
383             }
384             else
385             {
386                 y_start = 6;
387             }

```

```

388 for( xp = 0; xp < ap.PIXX; xp++ )
389     phase[xp][0] = 0.0;
390
391 y_start = 1;
392 first_time = 0;
393
394 for( yp = y_start; yp < n_scan_lines; yp++ )
395 {
396     if ( n_scans_read < ap.PIXY ) {
397         fread( wrapped_raster, sizeof( float ), ap.PIXX, wrapped_phase_map );
398         for( xp = 0; xp < ap.PIXX; xp++ )
399             phase[xp][yp] = wrapped_raster[ xp ];
400     }
401     if ( save_tan == 1 )
402         raster_tan[xp] = (unsigned short) (scale_tan*(phase[xp][yp]+t
403         an_offset)+0.5);
404
405     if ( save_tan == 1 )
406         TiffWriteScanline( out_tan, (u_char *) raster_tan,
407         (u_int) n_scans_read, (u_int) 0);
408     n_scans_read++;
409     else {
410         for( xp = 0; xp < ap.PIXX; xp++ )
411             phase[xp][yp] = 0.0;
412     }
413 }
414
415 void Compute_Phase_Row_Of_Tan_Fringes
416 (save_tan, save_mod, is_fringe_doubling)
417 int save_tan;
418 int save_mod;
419 int is_fringe_doubling;
420 {
421     int y_start;
422     int if;
423     if ( first_time != 1 )
424     {
425         int j;
426         for( i = 0; i < 6; i++ )
427             j = (ap.TILESIZEX+2) * -6 + i;
428         for( xp = 0; xp < ap.PIXX; xp++ )
429             phase[xp][i] = phase[xp][j];
430     }
431     y_start = 6;
432     else {
433         for( xp = 0; xp < ap.PIXX; xp++ )
434             phase[xp][0] = 0.0;
435     }
436     y_start = 1;
437     first_time = 0;
438 }

```

```

453 for( yp = y_start; yp < n_scan_lines; yp++ )
454 {
455     if ( n_scans_read < ap.PIXY ) {
456         for( i = 0; i < 3; i++ )
457             Read_Scanline( in[i], raster[i], n_scans_read );
458         for( xp = 0; xp < ap.PIXX; xp++ )
459         {
460             if ( in[0]->if_dir.td_bitspersample == 8 )
461             {
462                 IO = (double) raster[0][xp];
463                 Itheta = (double) raster[1][xp];
464                 I2theta = (double) raster[2][xp];
465             }
466             if ( in[0]->if_dir.td_bitspersample == 16 )
467             {
468                 IO = (double) ((unsigned short*) raster[0])[xp];
469                 Itheta = (double) ((unsigned short*) raster[1])[xp];
470                 I2theta = (double) ((unsigned short*) raster[2])[xp];
471             }
472             I2theta_minus_IO = I2theta - IO;
473             Itheta_minus_IO = Itheta - IO;
474             top = (I2theta_minus_IO * cos_2theta_minus_1) -
475                 (Itheta_minus_IO * cos_2theta_minus_1);
476             bottom = (sin_2theta * I2theta_minus_IO) -
477                 (sin_2theta * Itheta_minus_IO);
478             if ( is_fringe_doubling == 1 )
479             {
480                 if ( bottom == 0.0 )
481                     phase[xp][yp] = (float) PIF; /* 2 times pi/2 */
482                 else
483                     phase[xp][yp] = (float) (2.0*atan(top/bottom));
484             }
485             else {
486                 if ( bottom == 0.0 )
487                     phase[xp][yp] = (float) PIF;
488                 else
489                     phase[xp][yp] = (float) (atan2(top,bottom));
490             }
491             if ( save_tan == 1 )
492                 raster_tan[xp] =
493                     (unsigned short) (scale_tan*(phase[xp][yp]+tan_offset)+0.5
494 );
495             if ( save_mod == 1 )
496                 raster_mod[xp] = (unsigned char) 0;
497             if ( in[0]->if_dir.td_bitspersample == 8 )
498                 if ( ( fabs(top) < critical_size ) &&
499                     ( fabs(bottom) < critical_size ) )
500                 {
501                     phase[xp][yp] = DEAD_PIXEL;
502                     if ( save_mod == 1 )
503                         raster_mod[xp] = (unsigned char) 255;
504                 }
505             }
506 }

```

```

518 if ( in[0]>xtif_dlr.td bitpersample == 16 )
519 {
520 if ( ( fabs(top) < ( critical_size * 256.0 ) ) &&
521 ( fabs(bottom) < ( critical_size * 256.0 ) ) ) {
522 phase[xp][yp] = DEAD_PIXEL;
523 }
524 if ( save_mod == 1 )
525 raster_mod[xp] = (unsigned char) 255;
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 if ( save_tan == 1 )
534 TIFFWriteScanline( out_tan, (u_char *) raster_tan,
535 (u_int) n_scans_read, (u_int) 0);
536 }
537 }
538 if ( save_mod == 1 )
539 TIFFWriteScanline( out_mod, (u_char *) raster_mod,
540 (u_int) n_scans_read, (u_int) 0);
541 }
542 n_scans_read++;
543 }
544 else { for( xp = 0; xp < ap.PIXX; xp++ )
545 phase[xp][yp] = 0.0;
546 }
547 }
548 }
549 }
550 }
551 void Close_Phase_Step_Tan_Fringe_Computation
552 { save_tan, save_mod, is_fringe_doubling;
553 int save_tan;
554 int save_mod;
555 int is_fringe_doubling;
556 }
557 while( n_scans_read < ap.PIXY )
558 Compute_Phase_Step_Row_Of_Tan_Fringes
559 (save_tan, save_mod, is_fringe_doubling);
560 for( i = 0; i < 3; i++ )
561 {
562 TIFFClose( in[i] );
563 Free( raster[i] );
564 }
565 if ( save_tan == 1 ) {
566 TIFFFlushData( out_tan );
567 TIFFClose( out_tan );
568 Free( raster_tan );
569 }
570 if ( save_mod == 1 ) {
571 TIFFFlushData( out_mod );
572 TIFFClose( out_mod );
573 Free( raster_mod );
574 }
575 Free_Phase_Array( phase, ap.PIXX );
576 }
577 #define SNAP(a,b) tempr=(a); (a)=(b); (b)=tempr
578
583

```

```

584 void Fourn(data, nn, ndim, isign)
585 float data[];
586 int nn[], ndim, isign;
587 {
588 int i1, i2, i3, i2rev, i3rev, ip1, ip2, ip3, ifp1, ifp2;
589 int ibit, idim, k1, k2, n, nprev, nrem, ntot;
590 float tempi, temp2;
591 double theta, wi, wpi, wpr, wr, wtemp;
592
593 ntot=1;
594 for (idim=1; idim<ndim; idim++)
595 ntot *= nn[idim];
596 nprev=1;
597 for (idim=ndim; idim>1; idim--) {
598 nrem=ntot;
599 nrem=ntot/(n*nprev);
600 ip2=ip1*n;
601 ip3=ip2*nrem;
602 i2rev=i2;
603 i3rev=i3;
604 for (i2=1; i2<=ip2; i2+=ip1) {
605 if (i2 < i2rev) {
606 for (i1=i2; i1<=i2+ip1-2; i1+=2) {
607 for (i3=i1; i3<=ip3; i3+=ip2) {
608 i3rev=i2rev+i3-i2;
609 SNAP(data[i3], data[i3rev]);
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }

```

```

650 }
651
652 #undef SNAP
653
654 void Multiply_Scan_By_Filter_Window( scan, window )
655 float* scan;
656 float* window;
657 {
658     int k;
659     int i;
660     int i;
661     i = 1;
662
663     for( k = 0; k < n_padded_scan; k++ )
664     {
665         scan[i] *= window[k]; i += 2;
666     }
667
668     void Write_Scan_Data( fft_data, output_file )
669     float* fft_data;
670     char* output_file;
671 {
672     FILE* out;
673     int x;
674     int i;
675     int i;
676
677     out = fopen( output_file, "w" );
678
679     if ( out != NULL )
680     {
681         i = 1;
682         for( x = 0; x < n_padded_scan; x++ )
683         {
684             fprintf( out, "%f\n", fft_data[i] );
685             i += 2;
686         }
687         fclose( out );
688     }
689     else { Fpan_Error( frn_module, "Couldn't open scan output file\n" ); }
690 }
691
692 float Compute_Peak_Frequency( fft_data )
693 float* fft_data;
694 {
695     int x;
696     int i, j;
697     float magnitude;
698     float peak_frequency;
699     float peak_power;
700     int ncf;
701
702     i = 1;
703     for( x = 0; x < n_padded_scan; x++ )
704     {
705         j = i+1;
706
707         powers[x] = (float) sqrt( (double) ( fft_data[i]*fft_data[i] +
708                                             fft_data[j]*fft_data[j] ) );
709         i += 2;
710     }
711
712     peak_power = 0.0; ncf = 0;
713
714
715

```

```

716     for( x = 5; x < (n_padded_scan/2); x++ )
717     {
718         if ( powers[x] > peak_power ) { peak_power = powers[x]; ncf = x; }
719     }
720     peak_frequency = ((float) ncf)/((float) n_padded_scan);
721
722     if ( tile_rasters_completed == 0 ) {
723         printat( 2, 25 );
724     }
725
726     prn( "Raster %u, Peak Power at : %.5f cycles/pixel\n",
727          n_scans_read, peak_frequency );
728
729     }
730
731 #ifdef unixc
732     refresh();
733 #endif
734     return( peak_frequency );
735 }
736
737 void Write_FFT_Data( output_file )
738 char* output_file;
739 {
740     FILE* out;
741     int x;
742
743     out = fopen( output_file, "w" );
744
745     if ( out != NULL )
746     {
747         for( x = (n_padded_scan/2); x < (n_padded_scan); x++ )
748             fprintf( out, "%f\n", powers[x] );
749
750         for( x = 0; x < (n_padded_scan/2); x++ )
751             fprintf( out, "%f\n", powers[x] );
752
753         fprintf( out, "\n" );
754         fclose( out );
755     }
756     else { Fpan_Error( frn_module, "Couldn't open fft output file\n" ); }
757 }
758
759 void Compute_Filter_Window( window, n, nmax )
760 float* window;
761 double n;
762 unsigned nmax;
763 {
764     int k;
765     int i;
766     float wk;
767
768     for( i = 0; i < nmax; i++ )
769     {
770         k = i - (nmax/2);
771
772         wk = (float)
773         {
774             (
775                 1.0 +
776                 cos( (double) (PIF * ((double) k) / n ) )
777             )
778             / 2.0
779         };
780     }
781

```

```

782 window[i] = (float) sqrt( (double) wk);
783 }
784 }
785 }
786 }
787 void Compute_Trig_Sum_Window( window, sigma_coefs, n_sigma_coefs, N )
788 {
789     float* window;
790     float* sigma_coefs;
791     unsigned int n_sigma_coefs;
792     unsigned N;
793     {
794         unsigned t, k;
795         float sum;
796         double omega_t;
797         double K;
798         unsigned T;
799         T = N - 1;
800         for( t = 0 ; t <= T ; t++ )
801         {
802             sum = sigma_coefs[0];
803             for( k = 1; k < n_sigma_coefs; k++ )
804             {
805                 omega_t =
806                     ( (double) t ) * ( (double) k ) * ( (double) PIF )
807                     / ( (double) T );
808                 K = ( (double) k ) * ( (double) PIF ) / 2.0;
809                 sum += sigma_coefs[ k ] * ( float ) cos( omega_t - K );
810             }
811             window[ t ] = sum;
812         }
813     }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 void Write_Window_Data( window, nwindow, output_file )
822 {
823     FILE* window;
824     unsigned nwindow;
825     char* output_file;
826     {
827         FILE* out;
828         unsigned i;
829         out = fopen( output_file, "w" );
830         if ( out != NULL )
831         {
832             for( i = 0 ; i < nwindow; i++ )
833                 fprintf( out, "%f\n", window[i] );
834             fprintf( out, "\n" );
835             fclose( out );
836         }
837         else {
838             Fran_Error( fran_module, "Couldn't open window output file\n" );
839         }
840     }
841     void Initialise_FFT_Tan_Fringe_Computation( save_tan, save_mod, save_window, pref )
842     {
843         int save_tan;
844         int save_mod;
845         int save_window;
846     }
847 }

```

```

848 char* pref;
849 {
850     char string[256];
851     double r;
852     unsigned n;
853     unsigned i;
854     unsigned xp;
855     strcpy( prefix, pref );
856     scale_tan = 65535.0/(2.0*PIF);
857     tan_offset = PIF;
858     critical_size = 10.0;
859     n_scan_lines = (int) (ap.TILESIZE + 2);
860     in[0] = TIFFOpen( (char*) ap.file_names[0], "r" );
861     if ( in[0] == NULL )
862     {
863         sprintf( string, "couldn't open image file: %s", ap.file_names[0] );
864         Fran_Error( fran_module, string );
865     }
866     /* TIFFPrintDirectory( in[0], stdout, 0, 0 ); */
867     ap.PIXY = (unsigned) in[0]->tif_dir.td_imagelength;
868     ap.PIXX = (unsigned) in[0]->tif_dir.td_imagewidth;
869     raster[0] = (unsigned char*)
870         Malloc( TIFFScanlineSize( in[0] ) );
871     if ( raster[0] == NULL )
872         Fran_Error( fran_module, "Ran out of memory trying to allocate
873         tif raster" );
874     ap.NTILEX = (ap.PIXX/ap.XSTEP);
875     if ( ( ap.PIXX % ap.XSTEP ) != 0 ) ap.NTILEX++;
876     ap.NTILEY = (ap.PIXY/ap.YSTEP);
877     if ( ( ap.PIXY % ap.YSTEP ) != 0 ) ap.NTILEY++;
878     phase = Create_Phase_Array( ap.PIXX, (unsigned) n_scan_lines );
879     threshold_results = Allocate_Integer_Array( (ap.NTILEX-1) );
880     ap.CRIT_THRES_VAL = ( 0.1 * (float) ( ap.XSTEP * ap.YSTEP ) );
881     /* The FFT scan length must be a power of two.
882     The next section computes the next power of
883     two greater than the scan length.
884     It is padded at the ends of the scan to build
885     it to this width. A window is applied over the scan data only,
886     not the padding */
887     r = ( log( (double) ap.PIXX ) / log( (double) 2.0 ) );
888     n = (unsigned) r;
889     if ( ( r - ((double) (int) r) ) > 0.0 ) n++;
890     n_padded_scan = (unsigned) ( pow( 2.0, (double) n ) + 0.5 );
891     n_padded_zeros = ( n_padded_scan - ap.PIXX ) / 2;
892 }

```



```

913 window_spatial = Allocate_Float_Array( n_padded_scan );
914 window_fourier = Allocate_Float_Array( n_padded_scan );
915 scan = Allocate_Float_Array( 2 * n_padded_scan + 1 );
916 powers = Allocate_Float_Array( n_padded_scan );
917
918 Compute_Trig_Sum_Window( window_spatial, sigma_coeffs_spatial,
919                          n_coeffs_spatial, n_padded_scan );
920
921 ap.ASPECT = (float) ( ln[0]->tif_dir.td_resolution /
922                      ln[0]->tif_dir.td_resolution );
923
924 first_time = 1; n_scans_read = 0;
925
926 printat( 24, 25 );
927
928 prn( "Image Size : X=%u, Y=%u Pixels ( Padded X=%u )\n", ap.PIXX, ap.PIXY,
929      n_padded_scan );
930
931 if ( save_tan == 1 )
932 {
933     sprintf( frame, "%s.tan", pref );
934     out_tan = TIFFOpen( frame, "w" );
935     if ( out_tan == NULL ) Fran_Error( fran_module, "Couldn't open wrapped ph
936 ase map file" );
937     Setup_TIFF_Header( out_tan, ln[0],
938                       "Unwrapped Phase Map Generated by FFT Method\n", 16 );
939     raster_tan = (unsigned short*) Malloc( TIFFScanlineSize( out_tan ) );
940
941     if ( raster_tan == NULL )
942         Fran_Error( fran_module, "Ran out of memory trying to allocate tif ra
943 ster" );
944     if ( save_mod == 1 )
945     {
946         raster_mod = (unsigned char*) Malloc( TIFFScanlineSize( ln[0] ) );
947         if ( raster_mod == NULL )
948             Fran_Error( fran_module, "Ran out of memory trying to allocate tif ra
949 ster" );
950         sprintf( frame, "%s.mod", pref );
951         out_mod = TIFFOpen( frame, "w" );
952         if ( out_mod == NULL ) Fran_Error( fran_module, "Couldn't open low modula
953 tion file" );
954         Setup_TIFF_Header( out_mod, ln[0],
955                           "Low Modulation Points\n", 8 );
956         if ( save_window == 1 )
957         {
958             prn( "Saving window...\n" );
959             sprintf( frame, "%s.wnd", pref );
960             Write_Window_Data( window_spatial, n_padded_scan, frame );
961         }
962         if ( ap.carrier_freq < 0.0 )

```

```

975 {
976     Read_Scanline( ln[0], raster[0], ap.carrier_raster );
977     i = 0;
978     for( xp = 0; xp < n_padded_zeros; xp++ )
979     {
980         scan[i] = 0.0; i++;
981         scan[i] = 0.0; i++;
982     }
983     for( xp = 0; xp < ap.PIXX; xp++ )
984     {
985         if ( ln[0]->tif_dir.td_bitspersample == 8 )
986         {
987             scan[i] = (float) raster[0][xp];
988         }
989         if ( ln[0]->tif_dir.td_bitspersample == 16 )
990         {
991             scan[i] = (float) raster[0][xp];
992         }
993         if ( ln[0]->tif_dir.td_bitspersample == 16 )
994         {
995             scan[i] = (float) ((unsigned short*) raster[0])[xp];
996             i++;
997             scan[i] = 0.0;
998             i++;
999         }
1000     }
1001     for( xp = (ap.PIXX-n_padded_zeros); xp < n_padded_scan; xp++ )
1002     {
1003         scan[i] = 0.0; i++;
1004         scan[i] = 0.0; i++;
1005     }
1006     Multiply_Scan_By_Filter_Window( scan, window_spatial );
1007     Perform_One_Dimensional_FFT_On_Scan_Line( scan );
1008     ap.carrier_freq = Compute_Peak_Frequency( scan );
1009     tile_rasters_completed = 0;
1010     void Perform_One_Dimensional_FFT_On_Scan_Line( scan )
1011     {
1012         float* scan;
1013         int nn[2];
1014         nn[1] = n_padded_scan;
1015         Four( scan, nn, 1 );
1016         void Invert_One_Dimensional_FFT_On_Scan_Line( scan )
1017         {
1018             float* scan;
1019             int nn[2];
1020             nn[1] = n_padded_scan;
1021             Four( scan, nn, 1, -1 );
1022         }
1023         void Locate_Cut_Off_Frequencies_By_Thresholding_Power_Spectrum

```


Fringe Analysis	./fran/computewrapped.c	Page 18
1106	int dump_to_file;	
1107	{	
1108	int x;	
1109	int fd;	
1110	int r1,i1,r2,i2;	
1111	int src_dest;	
1112	int side_lobe_cut_off_freq;	
1113	float df;	
1114	float window_weight;	
1115	int nwindow;	
1116	int nwindow_fourier_flag;	
1117	int lower_bound;	
1118	int upper_bound;	
1119	int window_fourier_flag = 1;	
1120	fd = (int) (((float) n_padded_scan) * carrier_frequency) + 0.5);	
1121		
1122	/* Routine called next fits a polynomial to	
1123	part of the power spectrum and looks for the minimum	
1124	point between the DC lobe and the side lobe */	
1125		
1126	/* side_lobe_cut_off_freq =	
1127	Locate Side Lobe Cut Off Freq(scan, (unsigned) 2, fd, dump_to_file); */	
1128		
1129	Locate Cut Off Frequencies By Thresholding Power Spectrum	
1130	(powers, fd, *lower_bound, *upper_bound);	
1131		
1132	nwindow = (upper_bound - lower_bound + 1);	
1133		
1134	Compute Trig_Sum_Window(window_fourier, sigma_coeffs_fourier,	
1135	n_coeffs_fourier, (unsigned) nwindow);	
1136		
1137	if (dump_to_file >= 0) {	
1138	FILE* out;	
1139	unsigned i;	
1140	printf("name, \"%s.xtu.dat\", ap.files_prefix, dump_to_file);	
1141	out = fopen(frame, "w");	
1142	if (out != NULL)	
1143	{	
1144	for (i = 0 ; i < (n_padded_scan / 2) ; i++)	
1145	fprintf(out, "%f\n", 0.0);	
1146	fprintf(out, "%f\n", 0.0);	
1147	for (i = lower_bound; i < upper_bound; i++)	
1148	fprintf(out, "%f\n", window_fourier[i-lower_bound]);	
1149	for (i = 0 ; i < lower_bound; i++)	
1150	fprintf(out, "%f\n", 0.0);	
1151	fclose(out);	
1152	else {	
1153	Fran_Error(fran_module, "Couldn't open window output file\n"); }	
1154		
1155	for(x = (n_padded_scan/2); x < n_padded_scan; x++)	
1156	{	
1157	r1 = 1 + x * 2;	
1158		
1159		
1160		
1161		
1162		
1163		
1164		
1165		
1166		
1167		
1168		
1169		
1170		
1171		

Fringe Analysis	./fran/computewrapped.c	Page 17
1041	(powers, carrier_index, lower_bound, upper_bound)	
1042	float* powers;	
1043	int carrier_index;	
1044	int* lower_bound;	
1045	int* upper_bound;	
1046	{	
1047	int i,j;	
1048	int cut_off;	
1049	int checkband;	
1050	float power_threshold;	
1051	float sum_power;	
1052	checkband = (carrier_index / 3);	
1053		
1054	power_threshold = 1.0;	
1055	do {	
1056	*upper_bound = 0;	
1057	*lower_bound = 0;	
1058	sum_power = 0.0;	
1059	for(j = carrier_index; j > carrier_index-(checkband-1); j--)	
1060	sum_power += powers[j];	
1061	for(i = (carrier_index - (checkband-1)) ; i > 0; i--)	
1062	{	
1063	sum_power += powers[i];	
1064	if (sum_power < power_threshold) { *lower_bound = i; break; }	
1065	sum_power -= powers[i+(checkband-1)];	
1066	}	
1067	if (carrier_index + checkband < (n_padded_scan / 2))	
1068	{	
1069	sum_power = 0.0;	
1070	for(j = carrier_index; j < carrier_index+checkband-1; j++)	
1071	sum_power += powers[j];	
1072	for(i = (carrier_index + checkband-1) ;	
1073	i < (n_padded_scan / 2); i++)	
1074	{	
1075	sum_power += powers[i];	
1076	if (sum_power < power_threshold) { *upper_bound = i; break; }	
1077	sum_power -= powers[i-(checkband-1)];	
1078	}	
1079	else *upper_bound = ((n_padded_scan / 2) - 1);	
1080	power_threshold += ((float) checkband * 100.0);	
1081	while(*lower_bound == 0 *upper_bound == 0);	
1082	printf(
1083	"Carrier %d, Check Band = %d, Lower Bound = %d, Upper Bound = %d, Power = %10.0f	
1084	"\n", carrier_index, checkband, *lower_bound, *upper_bound, power_threshold);	
1085		
1086	int Shift_Side_Lobe_To_Origin(scan, carrier_frequency, dump_to_file)	
1087	float* scan;	
1088	float carrier_frequency;	
1089		
1090		
1091		
1092		
1093		
1094		
1095		
1096		
1097		
1098		
1099		
1100		
1101		
1102		
1103		
1104		
1105		

```

1172 il = ri+1;
1173 scan[r1] = 0.0;
1174 scan[i1] = 0.0;
1175 }
1176
1177 /* Filter Out Centre Lobe */
1178 for( x = 0; x < lower_bound; x++ )
1179 {
1180     r1 = 1 + x * 2;
1181     i1 = ri+1;
1182     scan[r1] = 0.0;
1183     scan[i1] = 0.0;
1184 }
1185
1186 for( src = 0; src < (n_padded_scan/2); src++ )
1187 {
1188     dest = src - fd;
1189     if ( dest < 0 ) dest += (n_padded_scan);
1190     if ( window_fourier_flag == 1 )
1191     {
1192         if ( (src < lower_bound) || (src > upper_bound) )
1193             window_weight = 0.0;
1194         else window_weight = window_fourier[ src - lower_bound ];
1195     }
1196     else window_weight = 1.0;
1197     r1 = 1 + src * 2;
1198     i1 = ri+1;
1199     r2 = 1 + dest * 2;
1200     i2 = ri+1;
1201     scan[r2] = scan[r1] * window_weight;
1202     scan[i2] = scan[i1] * window_weight;
1203 }
1204
1205 for( x = ((n_padded_scan/2)-fd); x < (n_padded_scan/2); x++ )
1206 {
1207     r1 = 1 + x * 2;
1208     i1 = ri+1;
1209     scan[r1] = 0.0;
1210     scan[i1] = 0.0;
1211 }
1212
1213 return( lower_bound );
1214
1215 void Compute_FFT_Row_Of_Tan_Fringes
1216 {
1217     save_tan_save_mod, is_fringe_doubling;
1218     int save_tan;
1219     int is_fringe_doubling;
1220     int y_start;
1221     int i, j;
1222     double real_imag;
1223     char fname[256];
1224     int side_lobe_cut;

```

```

1238 int dump_to_file = -1;
1239 for( i = 0; i < ap.NTILESX; i++ ) threshold_results[i] = 0;
1240 if ( first_time != 1 )
1241 {
1242     int j;
1243     for( i = 0; i < 6; i++ )
1244     {
1245         j = (ap.TILESIZE+2) * -6 + i;
1246         for( xp = 0; xp < ap.PIXX; xp++ )
1247             phase[xp][i] = phase[xp][j];
1248     }
1249     y_start = 6;
1250 }
1251 else
1252 {
1253     for( xp = 0; xp < ap.PIXX; xp++ )
1254         phase[xp][0] = 0.0;
1255     y_start = 1;
1256     first_time = 0;
1257 }
1258 for( yp = y_start; yp < n_scan_lines; yp++ )
1259 {
1260     if ( n_scans_read < ap.PIXY ) {
1261         dump_to_file = -1;
1262         if ( (dump_ffts_begin_scan <= n_scans_read) &&
1263             (dump_ffts_end_scan >= n_scans_read) )
1264             dump_to_file = n_scans_read;
1265         i = 1;
1266         Read_Scanline( in[0], raster[0], n_scans_read );
1267         for( xp = 0; xp < n_padded_zeros; xp++ )
1268             scan[i] = 0.0; i++;
1269             scan[i] = 0.0; i++;
1270     }
1271     for( xp = 0; xp < ap.PIXX; xp++ )
1272     {
1273         if ( in[0]->tif_dir.tif_bitspersample == 8 )
1274             scan[i] = (float) raster[0][xp];
1275         if ( scan[i] > THRESHOLD )
1276             threshold_results[xp/ap.XSTEP]++;
1277     }
1278     if ( in[0]->tif_dir.tif_bitspersample == 16 )
1279         scan[i] = (float) ((unsigned short*)raster[0])[xp];
1280     if ( scan[i] > (THRESHOLD * 256) )
1281         threshold_results[xp/ap.XSTEP]++;
1282     i++;
1283 }

```

```

1304 scan[i] = 0.0;
1305 i++;
1306 }
1307 for( xp = (ap.PIX*n_padded_zeros); xp < n_padded_scan; xp++ )
1308 {
1309     scan[i] = 0.0; i++;
1310     scan[i] = 0.0; i++;
1311 }
1312 if ( dump_to_file >= 0 )
1313 {
1314     printf( fname, "%s.r%.u.dat", prefix, n_scans_read );
1315     Write_Scan_Data( scan, fname );
1316 }
1317 Multiply_Scan_By_Filter_Window( scan, window_spatial );
1318 if ( dump_to_file >= 0 )
1319 {
1320     printf( fname, "%s.r%.u.dat", prefix, n_scans_read );
1321     Write_Scan_Data( scan, fname );
1322 }
1323 Perform_One_Dimensional_FFT_On_Scan_Line( scan );
1324 (void) Compute_Peak_Frequency( scan );
1325 if ( dump_to_file >= 0 )
1326 {
1327     printf( fname, "%s.f%.u.dat", prefix, n_scans_read );
1328     Write_FFT_Data( fname );
1329 }
1330 side_lobe_cut =
1331     Shift_Side_Lobe_To-Origin( scan, ap.carrier_freq, dump_to_file );
1332 if ( dump_to_file >= 0 )
1333 {
1334     FILE* ct;
1335     printf( fname, "%s.t%.u.dat", prefix, n_scans_read );
1336     for( xp = 0; xp < n_padded_scan; xp++ )
1337     {
1338         j = i+1;
1339         powers[xp] = (float) sqrt( (double) ( scan[i]*scan[i] +
1340                                             scan[j]*scan[j] ) );
1341         i += 2;
1342     }
1343     Write_FFT_Data( fname );
1344     printf( fname, "%s.ct%.u.dat", prefix, n_scans_read );
1345     ct = fopen( fname, "w" );
1346     if ( ct != NULL )
1347     {
1348         fprintf( ct, "%d\n", side_lobe_cut );
1349         fclose( ct );
1350     }
1351 }
1352 }

```

```

1370 Invert_One_Dimensional_FFT_On_Scan_Line( scan );
1371 for( xp = 0; xp < ap.PIX; xp++ )
1372 {
1373     i = 2 * (xp+n_padded_zeros) + 1;
1374     j = i+1;
1375     real = scan[i];
1376     imag = scan[j];
1377     top = imag;
1378     bottom = real;
1379     if ( is_fringe_doubling == 1 )
1380     {
1381         if ( bottom == 0.0 )
1382             phase[xp][yp] = (float) PIF; /* 2 times pi/2 */
1383         else
1384             phase[xp][yp] = (float) ( 2.0*atan(top/bottom) );
1385     }
1386     else
1387     {
1388         if ( bottom == 0.0 )
1389             phase[xp][yp] = (float) PIF;
1390         else
1391             phase[xp][yp] = (float) ( atan2(top,bottom) );
1392     }
1393     if ( save_tan == 1 )
1394         raster_tan[xp] = (unsigned short)
1395             (scale_tan*(phase[xp][yp]+tan_offset)+0.5);
1396 }
1397 if ( save_mod == 1 )
1398     raster_mod[xp] = (unsigned char) 0;
1399 if ( in[0] >= tif_dir.td_bitspersample == 8 )
1400 {
1401     if ( ( fabs(top) < critical_size ) &&
1402           ( fabs(bottom) < critical_size ) )
1403         phase[xp][yp] = DEAD_PIXEL;
1404     if ( save_mod == 1 )
1405         raster_mod[xp] = (unsigned char) 255;
1406 }
1407 if ( in[0] >= tif_dir.td_bitspersample == 16 )
1408 {
1409     if ( ( fabs(top) < (critical_size * 256.0) ) &&
1410           ( fabs(bottom) < (critical_size * 256.0) ) )
1411         phase[xp][yp] = DEAD_PIXEL;
1412     if ( save_mod == 1 )
1413         raster_mod[xp] = (unsigned char) 255;
1414 }
1415 }
1416 }
1417 }
1418 }
1419 }
1420 }
1421 }
1422 }
1423 }
1424 }
1425 }
1426 }
1427 }
1428 }
1429 }
1430 }
1431 }
1432 }
1433 }
1434 }

```

```

1435 if ( save_mod == 1 )
1436     TIFFWriteScanline( out_mod, (u_char *) raster_mod,
1437         (u_int) n_scans_read, (u_int) 0);
1438     n_scans_read++;
1439 }
1440 else {
1441     for( xp = 0; xp < ap.PIXX; xp++ )
1442         phase[xp][yp] = 0.0;
1443 }
1444 }
1445 }
1446 }
1447 }
1448 }
1449 }
1450 void Close_FFT_Tan_Fringe_Computation
1451 (save_tan_save_mod, is_fringe_doubling)
1452 {
1453     int save_tan;
1454     int save_mod;
1455     int is_fringe_doubling;
1456     {
1457         tile_rasters_completed = 1;
1458         while( n_scans_read < ap.PIXY )
1459             Compute_FFT_Row_Of_Tan_Fringes( save_tan, save_mod, is_fringe_doubling );
1460     }
1461     TIFFClose( in[0] );
1462     Free( raster[0] );
1463     if ( save_tan == 1 ) {
1464         TIFFFlushData( out_tan );
1465         TIFFClose( out_tan );
1466         Free( raster_tan );
1467     }
1468 }
1469 }
1470 if ( save_mod == 1 ) {
1471     TIFFFlushData( out_mod );
1472     TIFFClose( out_mod );
1473     Free( raster_mod );
1474 }
1475 }
1476 Free_Phase_Array( phase, ap.PIXX );
1477 }
1478 Free( (char*) threshold_results );
1479 Free( (char*) window_spatial );
1480 Free( (char*) window_fourier );
1481 Free( (char*) scan );
1482 }
1483 }
1484 void Initialise_Wrapped_Computation( save_tan, details_fname, pref )
1485 {
1486     int save_tan;
1487     char* details_fname;
1488     char* pref;
1489     {
1490         char string[256];
1491         double t;
1492         unsigned n;
1493         unsigned i;
1494         unsigned xp;
1495         int look_dot;
1496         FILE* in_file;
1497         char temp_fname[256];
1498         char* original_description;
1499         short bitspersample;
1500     }

```

```

1501     in_file = fopen( details_fname, "rb" );
1502     if ( in_file != NULL )
1503     {
1504         ap.PIXX = (unsigned) gets( in_file );
1505         ap.PIXY = (unsigned) gets( in_file );
1506         #ifdef unix
1507             refresh();
1508         #endif
1509         fclose( in_file );
1510     }
1511     else
1512         Fran_Error( fran_module, "Couldn't open wrapped phase map .fps file!");
1513     wrapped_raster = (float*) Malloc( ( sizeof(float) * ap.PIXX ) );
1514     if ( wrapped_raster == NULL )
1515         Fran_Error( fran_module, "Unable to allocate memory for wrapped raster!");
1516     wrapped_phase_map = fopen( ap.file_names[0], "rb" );
1517     if ( wrapped_phase_map == NULL )
1518         Fran_Error( fran_module, "Couldn't open wrapped phase map file!");
1519     sprintf( temp_fname, "%s.tif", ap.file_names[0] );
1520     strcpy( prefix, pref );
1521     scale_tan = 65535.0/(2.0*PIF);
1522     tan_offset = PIF;
1523     critical_size = 10.0;
1524     n_scan_lines = (int) (ap.TILSIZE + 2);
1525     in[0] = TIFFOpen( temp_fname, "w" );
1526     if ( in[0] == NULL )
1527     {
1528         sprintf( string, "Couldn't open image file: %s", temp_fname );
1529         Fran_Error( fran_module, string );
1530     }
1531     width = ap.PIXX;
1532     height = ap.PIXY;
1533     original_description = "Wrapped Phase Map\n";
1534     depth = 8;
1535     samplesperpixel = 1;
1536     bitspersample = 8;
1537     photometric = PHOTOMETRIC_MINISBLACK;
1538     bpsl = ( (long) width * (long) bitspersample * (long) samplesperpixel ) /
1539         8L;
1540     rowsperstrip = ( 8096L / bpsl );
1541     xres = (double) 72.0;
1542     yres = (double) 72.0;
1543     TIFFSetField( in[0], TIFFTAG_IMAGEWIDTH, width );
1544     TIFFSetField( in[0], TIFFTAG_IMAGELENGTH, height );
1545     TIFFSetField( in[0], TIFFTAG_BITSPERSAMPLE, bitspersample );
1546     TIFFSetField( in[0], TIFFTAG_COMPRESSION, COMPRESSION_NONE );

```

```

1567 TIFFSetField( in[0], TIFFTAG_PHOTOMETRIC, photometric );
1568 sprintf( document, "Project File : %s", ap.config_fname );
1569 TIFFSetField( in[0], TIFFTAG_DOCUMENTNAME, document );
1570
1571 #ifdef turbo
1572 Sec Date Time();
1573 TIFFSetField( in[0], TIFFTAG_DATETIME, date_time );
1574 TIFFSetField( in[0], TIFFTAG_HOSTCOMPUTER, "IBM PC or Compatible" );
1575 #endif
1576
1577 #ifdef unix
1578 void Close_Wrapped_Computation( save_tan )
1579 {
1580     int save_tan;
1581     char temp_fname[256];
1582     while( n_scans_read < ap.PIXX )
1583         Compute_Wrapped_Row_Of_Tan_Fringes( save_tan );
1584     TIFFClose( in[0] );
1585     sprintf( temp_fname, "%s.tif", ap.file_names[0] );
1586     unlink( temp_fname );
1587     fclose( wrapped_phase_map );
1588     if ( save_tan == 1 ) {
1589         TIFFFlushData( out_tan );
1590         TIFFClose( out_tan );
1591         Free( raster_tan );
1592     }
1593     Free_Phase_Array( phase, ap.PIXX );
1594
1595     void Write_Mesh_And_Grey_Scale_Map( grey_name, mesh_name,
1596         char* grey_name,
1597         char mesh_name,
1598         float gridx,
1599         float gridy,
1600         float minimum_height, maximum_height );
1601     int use_raster;
1602     int x,y;
1603     float xi,yi;
1604     int i;
1605     int tx,ty;
1606     int ox,oy;
1607     float h;
1608     int c;
1609     TIFF* out_grey;
1610     TIFF* out_mesh;
1611     FILE* f;
1612     int ty_last;
1613     unsigned char* raster_grey;
1614     printf( "Minimum Height %.5f, Maximum Height %.5f\n", minimum_height, maximum_height );
1615     out_grey = TIFFOpen( grey_name, "w" );
1616     ty_last = -1;
1617
1618     if ( out_tan == NULL ) Frn_Error( frn_module, "Couldn't open wrapped ph
1619 ase map file" );
1620
1621     Setup TIFF Reader( out_tan, in[0],
1622         "Unwrapped Phase Map Generated by FFT Method\n", 16 );
1623     raster_tan = (unsigned short*) Malloc( TIFFScanlineSize( out_tan ) );
1624     if ( raster_tan == NULL )

```

```

1631     }
1632     Frn_Error( frn_module, "Ran out of memory trying to allocate tif ra
1633 ster" );
1634
1635     tile_rasters_completed = 0;
1636
1637 #ifdef unix
1638     refresh();
1639 #endif
1640
1641 void Close_Wrapped_Computation( save_tan )
1642 {
1643     int save_tan;
1644     char temp_fname[256];
1645     while( n_scans_read < ap.PIXX )
1646         Compute_Wrapped_Row_Of_Tan_Fringes( save_tan );
1647     TIFFClose( in[0] );
1648     sprintf( temp_fname, "%s.tif", ap.file_names[0] );
1649     unlink( temp_fname );
1650     fclose( wrapped_phase_map );
1651     if ( save_tan == 1 ) {
1652         TIFFFlushData( out_tan );
1653         TIFFClose( out_tan );
1654         Free( raster_tan );
1655     }
1656     Free_Phase_Array( phase, ap.PIXX );
1657
1658     void Write_Mesh_And_Grey_Scale_Map( grey_name, mesh_name,
1659         char* grey_name,
1660         char mesh_name,
1661         float gridx,
1662         float gridy,
1663         float minimum_height, maximum_height );
1664     int use_raster;
1665     int x,y;
1666     float xi,yi;
1667     int i;
1668     int tx,ty;
1669     int ox,oy;
1670     float h;
1671     int c;
1672     TIFF* out_grey;
1673     TIFF* out_mesh;
1674     FILE* f;
1675     int ty_last;
1676     unsigned char* raster_grey;
1677     printf( "Minimum Height %.5f, Maximum Height %.5f\n", minimum_height, maximum_height );
1678     out_grey = TIFFOpen( grey_name, "w" );
1679     ty_last = -1;
1680
1681     if ( out_tan == NULL ) Frn_Error( frn_module, "Couldn't open wrapped ph
1682 ase map file" );
1683
1684     Setup TIFF Reader( out_tan, in[0],
1685         "Unwrapped Phase Map Generated by FFT Method\n", 16 );
1686     raster_tan = (unsigned short*) Malloc( TIFFScanlineSize( out_tan ) );
1687     if ( raster_tan == NULL )

```

Fringe Analysis	./fran/computewrapped.c	Page 27
1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759	<pre>if (out_grey != NULL) { raster_grey = (unsigned char*) Malloc(TIFFScanlineSize(in[0])); if (raster_grey == NULL) Fran_Error(fran_module, "Ran out of memory trying to allocate tif ras ter"); } Setup TIFF Header(out_grey, in[0], "ndgrey Scale Representation of Unwrapped Phase\n", 8); out_mesh = fopen(mesh_name, "w"); if (out_mesh != NULL) { x1 = y1 = 0.0; in_mesh = 1; for(y = 0; y < ap.PIXY; y++) { for(x = 0; x < ap.PIXX; x++) { if (x1 >= ((float) ap.PIXX)) { x1 = 0.0; y1 += gridx; fprintf(out_mesh, "\n"); } if ((((int) (y1 + 0.5)) == y) && (((int) (x1 + 0.5)) == x)) { in_mesh = 1; x1 += gridx; } else in_mesh = 0; tx = x / (ap.XSTEP); ox = x % (ap.XSTEP); ty = y / (ap.YSTEP); oy = y % (ap.YSTEP); if (use_ram == 0) { if ((ty != ty_last) && (ty_last >= 0) && (ty_last < ap.NTILESY)) for(i = 0; i < ap.NTILESX; i++) if ((t = ta[i][ty_last]) != NULL) if (t->status >= TILE_OFFSET_COMPUTED) { FreeHeightArray(t->tile, (ap.TILESIZE-4)); t->tile = NULL; } } ty_last = ty; } if ((tx >= ap.NTILESX) (ty >= ap.NTILESY) ((t = ta[tx][ty]) == NULL)) { raster_grey[x] = (unsigned char) 0; if (in_mesh == 1) fprintf(out_mesh, "%f", minimum_height); } } } }</pre>	
Fringe Analysis	./fran/computewrapped.c	Page 28
1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812	<pre> } else { if (t->status >= TILE_OFFSET_COMPUTED) { if (use_ram == 0) && (t->tile == NULL)) { t->tile = ReadHeightArray(t->start, (ap.TILESIZE-4), (ap.TILESIZE-4), ap.tile_solutions); if (t->tile[ox][oy] < LIMIT_HEIGHT) h = t->offset+t->tile[ox][oy]; else h = minimum_height; if (in_mesh == 1) fprintf(out_mesh, "%f", h); h = ((h - minimum_height) / (maximum_height - minimum_height)); c = (unsigned char) (255.0*h); raster_grey[x] = c; } else { raster_grey[x] = (unsigned char) 0; if (in_mesh == 1) fprintf(out_mesh, "%f", minimum_height); } } } TIFFWriteScanline(out_grey, (u_char *) raster_grey, (u_int) y, (u_int) 0); } fclose(out_mesh); } else { Fran_Error(fran_module, "Couldn't open mesh output file"); } TIFFFlushData(out_grey); TIFFClose(out_grey); Free(raster_grey); } else { Fran_Error(fran_module, "Couldn't open grey scale map, output file"); } }</pre>	

```
1 /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  *
5  */
6 extern void Explore();
7 extern void Find_Min_And_Max_Heights();
8 extern void Write_State_File();
9
10 #define DIFF_NOT_SET (0.0)
11 #define EDGE_DOES_NOT_EXIST (-1)
```


Fringe Analysis	./fran/correctoffsets.c	Page 2
<pre> 67 switch(x) { 68 case TILE_ABOVE : if (ty != 0) 69 t2 = ta[tx][ty-1]; 70 break; 71 case TILE_BELOW : if (ty != (ap.NTILESX-1)) 72 t2 = ta[tx][ty+1]; 73 break; 74 case TILE_RIGHT : if (tx != (ap.NTILESY-1)) 75 t2 = ta[tx+1][ty]; 76 break; 77 case TILE_LEFT : if (tx != 0) 78 t2 = ta[tx-1][ty]; 79 break; 80 case TILE_NONE : if (tx != 0) 81 t2 = ta[tx-1][ty]; 82 break; 83 case TILE_NONE : if (tx != 0) 84 t2 = ta[tx-1][ty]; 85 break; 86 case TILE_NONE : if (tx != 0) 87 t2 = ta[tx-1][ty]; 88 break; 89 case TILE_NONE : if (tx != 0) 90 t2 = ta[tx-1][ty]; 91 break; 92 case TILE_NONE : if (tx != 0) 93 t2 = ta[tx-1][ty]; 94 break; 95 case TILE_NONE : if (tx != 0) 96 t2 = ta[tx-1][ty]; 97 break; 98 case TILE_NONE : if (tx != 0) 99 t2 = ta[tx-1][ty]; 100 break; 101 case TILE_NONE : if (tx != 0) 102 t2 = ta[tx-1][ty]; 103 break; 104 case TILE_NONE : if (tx != 0) 105 t2 = ta[tx-1][ty]; 106 break; 107 case TILE_NONE : if (tx != 0) 108 t2 = ta[tx-1][ty]; 109 break; 110 case TILE_NONE : if (tx != 0) 111 t2 = ta[tx-1][ty]; 112 break; 113 case TILE_NONE : if (tx != 0) 114 t2 = ta[tx-1][ty]; 115 break; 116 case TILE_NONE : if (tx != 0) 117 t2 = ta[tx-1][ty]; 118 break; 119 case TILE_NONE : if (tx != 0) 120 t2 = ta[tx-1][ty]; 121 break; 122 case TILE_NONE : if (tx != 0) 123 t2 = ta[tx-1][ty]; 124 break; 125 case TILE_NONE : if (tx != 0) 126 t2 = ta[tx-1][ty]; 127 break; 128 case TILE_NONE : if (tx != 0) 129 t2 = ta[tx-1][ty]; 130 break; 131 case TILE_NONE : if (tx != 0) 132 t2 = ta[tx-1][ty]; 133 break; </pre>	<pre> 67 switch(x) { 68 case TILE_ABOVE : if (ty != 0) 69 t2 = ta[tx][ty-1]; 70 break; 71 case TILE_BELOW : if (ty != (ap.NTILESX-1)) 72 t2 = ta[tx][ty+1]; 73 break; 74 case TILE_RIGHT : if (tx != (ap.NTILESY-1)) 75 t2 = ta[tx+1][ty]; 76 break; 77 case TILE_LEFT : if (tx != 0) 78 t2 = ta[tx-1][ty]; 79 break; 80 case TILE_NONE : if (tx != 0) 81 t2 = ta[tx-1][ty]; 82 break; 83 case TILE_NONE : if (tx != 0) 84 t2 = ta[tx-1][ty]; 85 break; 86 case TILE_NONE : if (tx != 0) 87 t2 = ta[tx-1][ty]; 88 break; 89 case TILE_NONE : if (tx != 0) 90 t2 = ta[tx-1][ty]; 91 break; 92 case TILE_NONE : if (tx != 0) 93 t2 = ta[tx-1][ty]; 94 break; 95 case TILE_NONE : if (tx != 0) 96 t2 = ta[tx-1][ty]; 97 break; 98 case TILE_NONE : if (tx != 0) 99 t2 = ta[tx-1][ty]; 100 break; 101 case TILE_NONE : if (tx != 0) 102 t2 = ta[tx-1][ty]; 103 break; 104 case TILE_NONE : if (tx != 0) 105 t2 = ta[tx-1][ty]; 106 break; 107 case TILE_NONE : if (tx != 0) 108 t2 = ta[tx-1][ty]; 109 break; 110 case TILE_NONE : if (tx != 0) 111 t2 = ta[tx-1][ty]; 112 break; 113 case TILE_NONE : if (tx != 0) 114 t2 = ta[tx-1][ty]; 115 break; 116 case TILE_NONE : if (tx != 0) 117 t2 = ta[tx-1][ty]; 118 break; 119 case TILE_NONE : if (tx != 0) 120 t2 = ta[tx-1][ty]; 121 break; 122 case TILE_NONE : if (tx != 0) 123 t2 = ta[tx-1][ty]; 124 break; 125 case TILE_NONE : if (tx != 0) 126 t2 = ta[tx-1][ty]; 127 break; 128 case TILE_NONE : if (tx != 0) 129 t2 = ta[tx-1][ty]; 130 break; 131 case TILE_NONE : if (tx != 0) 132 t2 = ta[tx-1][ty]; 133 break; </pre>	
Fringe Analysis	./fran/correctoffsets.c	Page 1
<pre> 1 /* Copyright (c) 1991 by Tom Judge. 2 * All rights reserved. 3 */ 4 5 #include <ctype.h> 6 #include <math.h> 7 #include <stdio.h> 8 #include <stdlib.h> 9 #include <string.h> 10 #include <sys/types.h> 11 #include <unistd.h> 12 #include <fcntl.h> 13 #include <sys/stat.h> 14 #include <sys/time.h> 15 #include <sys/types.h> 16 #include <sys/time.h> 17 #include <sys/types.h> 18 #include <sys/time.h> 19 #include <sys/types.h> 20 #include <sys/time.h> 21 #include <sys/types.h> 22 #include <sys/time.h> 23 #include <sys/types.h> 24 #include <sys/time.h> 25 #include <sys/types.h> 26 #include <sys/time.h> 27 #include <sys/types.h> 28 #include <sys/time.h> 29 #include <sys/types.h> 30 #include <sys/time.h> 31 #include <sys/types.h> 32 #include <sys/time.h> 33 #include <sys/types.h> 34 #include <sys/time.h> 35 #include <sys/types.h> 36 #include <sys/time.h> 37 #include <sys/types.h> 38 #include <sys/time.h> 39 #include <sys/types.h> 40 #include <sys/time.h> 41 #include <sys/types.h> 42 #include <sys/time.h> 43 #include <sys/types.h> 44 #include <sys/time.h> 45 #include <sys/types.h> 46 #include <sys/time.h> 47 #include <sys/types.h> 48 #include <sys/time.h> 49 #include <sys/types.h> 50 #include <sys/time.h> 51 #include <sys/types.h> 52 #include <sys/time.h> 53 #include <sys/types.h> 54 #include <sys/time.h> 55 #include <sys/types.h> 56 #include <sys/time.h> 57 #include <sys/types.h> 58 #include <sys/time.h> 59 #include <sys/types.h> 60 #include <sys/time.h> 61 #include <sys/types.h> 62 #include <sys/time.h> 63 #include <sys/types.h> 64 #include <sys/time.h> 65 #include <sys/types.h> 66 #include <sys/time.h> </pre>	<pre> 35 int tx,ty; 36 int tx2,ty2; 37 int tx3,ty3; 38 float o[4]; 39 int s[4]; 40 int count[4]; 41 int ncomputed; 42 int nname; 43 int ibest; 44 int i; 45 int changed; 46 changed = 0; 47 48 for(ty = 0; ty < ap.NTILESY; ty++) 49 for(tx = 0; tx < ap.NTILESX; tx++) 50 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 51 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 52 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 53 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 54 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 55 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 56 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 57 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 58 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 59 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 60 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 61 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 62 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 63 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 64 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 65 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) 66 if ((tx != 0 ty != 0) (tx == 0 ty == 0)) </pre>	

Fringe Analysis	./fran/correctoffsets.c	Page 3
133	!= ((int) (1000.0*t1->offset))	
134	}	
135	if (ncomputed != 4) {	
136	t1->offset = o[ibest];	
137	PrintOffset = o[ibest];	
138	}	
139	"Corrected on basis of 2 matching neighbours out of %d at tile %d,%d\n",	
140	ncomputed,tx,ty);	
141	changed = 1;	
142	}	
143	}	
144	break;	
145	case 3 :	
146	case 4 :	
147	{	
148	{ (int) (1000.0*(o[ibest]))	
149	!= ((int) (1000.0*t1->offset))	
150	{	
151	{ t1->offset = o[ibest];	
152	PrintOffset = o[ibest];	
153	}	
154	"Corrected on basis of %d matching neighbours out of %d at tile %d,%d\n",	
155	nname,ncomputed,tx,ty);	
156	changed = 1;	
157	}	
158	break;	
159	}	
160	{	
161	{	
162	{	
163	{	
164	{	
165	{	
166	{	
167	{	
168	{	
169	{	
170	{	
171	{	
172	{	
173	{	
174	{	
175	{	
176	{	
177	{	
178	{	
179	{	
180	{	
181	{	
182	{	
183	{	
184	{	
185	{	
186	{	
187	{	
188	{	
189	{	
190	{	
191	{	
192	{	
193	{	
194	{	
195	{	
196	{	
197	{	
198	{	

Fringe Analysis	./fran/correctoffsets.c	Page 4
199	if (tx != (ap.NTILESX-1))	
200	t2 = ta[(tx+1)][ty];	
201	break;	
202	case TILE_LEFT :	
203	if (tx != 0)	
204	t2 = ta[(tx-1)][ty];	
205	break;	
206	}	
207	}	
208	if (t2 != NULL)	
209	{	
210	if (t2->status == TILE_OFFSET_COMPUTED)	
211	{	
212	{	
213	{	
214	{	
215	{	
216	{	
217	{	
218	{	
219	{	
220	{	
221	{	
222	{	
223	{	
224	{	
225	{	
226	{	
227	{	
228	{	
229	{	
230	{	
231	{	
232	{	
233	{	
234	{	
235	{	
236	{	
237	{	
238	{	
239	{	
240	{	
241	{	
242	{	
243	{	
244	{	
245	{	
246	{	
247	{	
248	{	
249	{	
250	{	
251	{	
252	{	
253	{	
254	{	
255	{	
256	{	
257	{	
258	{	
259	{	
260	{	
261	{	
262	{	
263	{	
264	{	

```

265 for( x = 0; x < ap.NTILESX; x++)
266 {
267     if ( ta[x][y] != NULL )
268     {
269         if ( ta[x][y]->status >= TILE_OFFSET_COMPUTED )
270         {
271             min = ta[x][y]->offset + ta[x][y]->min_height;
272             max = ta[x][y]->offset + ta[x][y]->max_height;
273             if ( min > -LIMIT_HEIGHT && max < LIMIT_HEIGHT ) {
274                 if ( first_time == 1 )
275                 {
276                     *minimum_height = min;
277                     *maximum_height = max;
278                     first_time = 0;
279                 }
280                 else {
281                     if ( min < *minimum_height )
282                         *minimum_height = min;
283                     if ( max > *maximum_height )
284                         *maximum_height = max;
285                 }
286             }
287         }
288     }
289 }
290
291 void Write Stats_File( name )
292 char* name;
293 {
294     FILE* out;
295     long n_if_all;
296     float percentage_of_field_solved;
297     long temp;
298     long temp2;
299     long temp3;
300     long temp4;
301     long temp5;
302     long temp6;
303     long temp7;
304     long temp8;
305     temp = (long) ( ap.NTILESX );
306     temp2 = (long) ( ap.NTILESY );
307     temp3 = temp * temp2 * 4L;
308     temp4 = temp * temp2 * 2L;
309     temp5 = ( temp * temp2 ) * 2L;
310     temp6 = ( temp * temp2 ) * 2L;
311     n_if_all = temp3 - temp4;
312     percentage_of_field_solved =
313         ( 100.0 * ((float) (n_boundaries - mismatched_count))
314           / ((float) n_if_all) );
315     printf( "Total number of tile sides in complete frame = %ld\n", n_if_all );
316     printf( "Number of sides which computed between tiles = %ld\n",
317             n_boundaries );
318     printf( "Number of sides which did not appear to match = %ld\n",
319             mismatched_count );
320     printf( "Estimate of frame area solved = %.2f%%\n", percentage_of_field_solved );
321     out = fopen( name, "w" );
322 }
330

```

```

331 if ( out != NULL ) {
332     fprintf( out,
333             "Total number of tile sides in complete frame = %ld\n", n_if_all );
334     fprintf( out,
335             "Number of sides which computed between tiles = %ld\n",
336             n_boundaries );
337     fprintf( out,
338             "Number of sides which did not appear to match = %ld\n",
339             mismatched_count );
340     fprintf( out,
341             "Estimate of frame area solved = %.2f%%\n", percentage_of_field_solved );
342     fclose( out );
343 }
344 else printf( "Couldn't open statistics file: %s\n", name );
345 }
346 }
347 }

```

Fringe Analysis	./fran/dynamicarray.h	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5		
6	typedef TILE_ELEM* ELEM;	
7	typedef ELEM* SINGLE_DIM_ARRAY;	
8	typedef SINGLE_DIM_ARRAY ELEM*	
9	typedef ELEM* TILES;	
10		
11	extern TILES Create_Tile_Array();	
12	extern void Free_Tile_Array();	
13		
14	extern HEIGHTS Create_Height_Array();	
15	extern void Free_Height_Array();	
16		
17	extern BYTES Create_Byte_Array();	
18	extern void Free_Byte_Array();	
19		
20	extern PHASE Create_Phase_Array();	
21	extern void Free_Phase_Array();	
22		
23	extern OVERLAP Create_Overlap_Array();	
24	extern void Free_Overlap_Array();	
25		
26	extern FLOATS Create_Float_Array();	
27	extern void Free_Float_Array();	

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5
6  #include <stdio.h>
7  #include <math.h>
8  #include "franhed.h"
9  #define turboc
10 #include <alloc.h>
11 #endif
12 #define unixc
13 #include <alloc.h>
14 #include <urses.h>
15 #endif
16 #include "memory.h"
17 #include "dynamicarray.h"
18 #include "main.h"
19
20 TILES Create_Tile_Array( nx, ny )
21 unsigned nx,ny;
22 {
23     TILES tiles;
24     int i;
25
26     tiles = (TILES) Malloc((nx)*(sizeof(ELEMXX)));
27     if ( tiles != NULL )
28     {
29         for( i = 0; i < nx; i++)
30             tiles[i] = (ELEMXX) Calloc(ny, (sizeof(ELEMXX)));
31     }
32     if ( tiles[i] == NULL )
33     {
34         Franh_Error( franh_module, "Ran out of memory trying to allocate
35         array of tiles\n");
36     }
37 }
38
39 else
40 {
41     Franh_Error(franh_module,"Ran out of memory trying to allocate array of til
42     es\n");
43 }
44
45 return( tiles );
46 }
47
48 void Free_Tile_Array( tiles, nx )
49 TILES tiles;
50 unsigned nx;
51 {
52     int i;
53
54     if ( tiles != NULL )
55     {
56         for( i = 0; i < nx; i++)
57             Free( (char*) tiles[i] );
58         Free( (char*) tiles );
59     }
60 }
61
62 HEIGHTS Create_Height_Array( nx, ny )
63 unsigned nx,ny;

```

```

65 {
66     HEIGHTS heights;
67     int i;
68
69     heights = (HEIGHTS) Malloc(nx)*(sizeof(float));
70
71     if ( heights != NULL )
72     {
73         for( i = 0; i < nx; i++)
74         {
75             heights[i] = (float*) Calloc(ny, (sizeof(float)));
76
77             if ( heights[i] == NULL )
78                 Fran_Error( fran_module, "Ran out of memory trying to allocate
79 height array\n");
80         }
81     }
82     else Fran_Error( fran_module, "Ran out of memory trying to allocate height arr
83 ay\n");
84
85     return( heights );
86
87 void Free Height_Array( heights, nx )
88 HEIGHTS heights;
89 unsigned nx;
90 {
91     int i;
92
93     if ( heights != NULL )
94     {
95         for( i = 0; i < nx; i++)
96             Free( (char*) heights[i] );
97
98         Free( (char*) heights );
99     }
100
101     BYTES Create_Byte_Array( nx, ny )
102     unsigned nx, ny;
103 {
104     BYTES bytes;
105     int i;
106
107     bytes = (BYTES) Malloc(nx)*(sizeof(unsigned char));
108
109     if ( bytes != NULL )
110     {
111         for( i = 0; i < nx; i++)
112         {
113             bytes[i] = (unsigned char*) Calloc(ny, (sizeof(unsigned char)));
114
115             if ( bytes[i] == NULL )
116                 Fran_Error( fran_module, "Ran out of memory trying to allocate b
117 yte array\n");
118         }
119     }
120     else Fran_Error( fran_module, "Ran out of memory trying to allocate byte array\
121 n");
122
123     return( bytes );
124
125 void Free_Byte_Array( bytes, nx )
126 BYTES bytes;
127 unsigned nx;

```

Fringe Analysis	/fran/dynamicarray.c	Page 3
127	{	
128	int i;	
129	if (bytes != NULL)	
130	{	
131	for(i = 0; i < nx; i++)	
132	Free((char*) bytes[i]);	
133		
134	Free((char*) bytes);	
135	}	
136		
137		
138	PHASE Create_Phase_Array(nx, ny)	
139	unsigned nx, ny;	
140	{	
141	PHASE phase;	
142	int i;	
143		
144	phase = (PHASE) Malloc((nx)*(sizeof(float)));	
145		
146	if (phase != NULL)	
147	{	
148	for(i = 0; i < nx; i++)	
149	{	
150	phase[i] = (float*) Calloc(ny, (sizeof(float)));	
151		
152	if (phase[i] == NULL)	
153	{	
154	Free_Error(fran_module, "Ran out of memory trying to allocate p	
155	hase array\n");	
156	}	
157	else Free_Error(fran_module, "Ran out of memory trying to allocate phase array	
158	\n");	
159		
160	return(phase);	
161	}	
162	void Free_Phase_Array(phase, nx)	
163	PHASE phase;	
164	unsigned nx;	
165	{	
166	int i;	
167		
168	if (phase != NULL)	
169	{	
170	for(i = 0; i < nx; i++)	
171	Free((char*) phase[i]);	
172		
173	Free((char*) phase);	
174		
175	}	
176		
177	OVERLAP Create_Overlap_Array(nx, ny, nz)	
178	unsigned nx, ny, nz;	
179	{	
180	OVERLAP overlap;	
181	int i, j;	
182	overlap = (OVERLAP) Malloc((nx)*(sizeof(float**)));	
183		
184	if (overlap != NULL)	
185	{	
186	for(i = 0; i < nx; i++)	
187	{	
188	overlap[i] = (float**) Calloc(ny, (sizeof(float**)));	
189		
190		

Fringe Analysis	/fran/dynamicarray.c	Page 4
191	if (overlap[i] != NULL)	
192	{	
193	for(j = 0; j < ny; j++)	
194	{	
195	overlap[i][j] = (float*) Calloc(nz, (sizeof(float)));	
196		
197	if (overlap[i][j] == NULL)	
198	{	
199	Free((char*) overlap[i][j]);	
200	Free((char*) overlap[i]);	
201	Free((char*) overlap);	
202	return(overlap);	
203		
204	void Free_Overlap_Array(overlap, nx, ny)	
205	{	
206	OVERLAP overlap;	
207	unsigned nx, ny;	
208	{	
209	int i, j;	
210		
211	if (overlap != NULL)	
212	{	
213	for(i = 0; i < nx; i++)	
214	{	
215	for(j = 0; j < ny; j++)	
216	{	
217	Free((char*) overlap[i][j]);	
218		
219	Free((char*) overlap[i]);	
220		
221	Free((char*) overlap);	
222		
223		
224		
225		
226	Free((char*) overlap);	
227		
228		
229	floats Create_Float_Array(nx, ny)	
230	unsigned nx, ny;	
231	{	
232	floats floats;	
233	int i;	
234		
235	floats = (floats) Malloc((nx)*(sizeof(float**)));	
236		
237	if (floats != NULL)	
238	{	
239	for(i = 0; i < nx; i++)	
240	{	
241	floats[i] = (float*) Calloc(ny, (sizeof(float)));	
242		
243	if (floats[i] == NULL)	
244	{	
245	Free((char*) floats[i]);	
246	Free((char*) floats);	
247	return(floats);	
248		
249		
250		
251		

```
252 void Free Float_Array( floats, nx )
253 {
254     unsigned nx;
255     int i;
256     if ( floats != NULL )
257     {
258         for( i = 0; i < nx; i++ )
259             Free( (char*) floats[i] );
260         Free( (char*) floats );
261     }
262 }
```

Fringe Analysis	./fran/fringecount.h	Page 1
1	extern float Unwrap_Verical_Scans();	
2	extern float Unwrap_Horizontal_Scans();	
3	extern int* Allocate_Integer_Array();	
4	extern float* Allocate_Float_Array();	

Fringe Analysis	./fran/fringeCount.c	Page 2
67	if (theta < 0.0) /* i.e the step hasn't happened yet */	
68	{	
69	point_height = offset + theta;	
70	offset -= height_of_one_fringe;	
71	}	
72	else /* the step has happened */	
73	{	
74	offset -= height_of_one_fringe;	
75	point_height = offset + theta;	
76	}	
77	nfringe_edges++;	
78	on_edge = 1;	
79	}	
80	if (is_down != 0)	
81	{	
82	if (theta > 0.0) /* i.e the step hasn't happened yet */	
83	{	
84	point_height = offset + theta;	
85	offset += height_of_one_fringe;	
86	}	
87	else /* the step has happened */	
88	{	
89	offset += height_of_one_fringe;	
90	point_height = offset + theta;	
91	}	
92	nfringe_edges++; /* Yes : increment count of edges */	
93	on_edge = 1;	
94	}	
95	}	
96	/* End Of Step Routine */	
97	if (y == 0) start_of_scan_offset = point_height;	
98	point_height -= start_of_scan_offset;	
99	h[x][y] = point_height;	
100	num_fringes_passed[x] = nfringe_edges;	
101	confidence = (float)	
102	{ ap.TILESIZE*ap.TILESIZE - length_of_edges_parallel_to_vertical_scan_directio	
103	n };	
104	return(confidence);	
105	}	
106	float Unwrap Horizontal Scans(e, h, num_fringes_passed, tile_phase)	
107	BYTES e; /* Data array containing fringe edges */	
108	HEIGHTS h;	
109	int num_fringes_passed[];	
110	int num_fringes_passed[];	
111	int num_fringes_passed[];	
112	float point_height;	
113	float theta;	
114	int x,y;	
115	int nfringe_edges;	
116	float offset;	
117	int is_up;	
118	int is_down;	

Fringe Analysis	./fran/fringeCount.c	Page 1
1	#include <stdio.h>	
2	#include <math.h>	
3	#include "franhed.h"	
4	#define TURBOC	
5	#include <alloc.h>	
6	#endif	
7	#endif	
8	#include <memory.h>	
9	#include <conio.h>	
10	#include "memory.h"	
11	#include "dynamicarray.h"	
12	#include "main.h"	
13	#include "fringeCount.h"	
14	#include "fringeCount.h"	
15	float Unwrap Vertical Scans(e, h, num_fringes_passed, tile_phase)	
16	BYTES e; /* Data array containing fringe edges */	
17	HEIGHTS h;	
18	int num_fringes_passed[];	
19	int num_fringes_passed[];	
20	int num_fringes_passed[];	
21	float point_height;	
22	float theta;	
23	int x,y;	
24	int nfringe_edges;	
25	float offset;	
26	int is_up;	
27	int is_down;	
28	float height_of_one_fringe;	
29	float start_of_scan_offset;	
30	float confidence;	
31	int length_of_edges_parallel_to_vertical_scan_direction = 0;	
32	int edge_data;	
33	height_of_one_fringe = 2.0 * PI;	
34	for(x = 0; x < ap.TILESIZE; x++)	
35	{	
36	offset = 0.0; nfringe_edges = 0; on_edge = 0;	
37	for(y = 0; y < ap.TILESIZE; y++)	
38	{	
39	point_height = 0.0;	
40	theta = tile_phase[x+1][y+1];	
41	edge_data = (int) e[x][y];	
42	if (edge_data == 0)	
43	{	
44	on_edge = 0;	
45	point_height = offset + theta;	
46	}	
47	else {	
48	if (on_edge == 1)	
49	{	
50	length_of_edges_parallel_to_vertical_scan_direction++;	
51	point_height = offset + theta;	
52	}	
53	else	
54	{	
55	is_up = edge_data & STEP_UP_EDGE_TOP_TO_BOTTOM;	
56	is_down = edge_data & STEP_DOWN_EDGE_TOP_TO_BOTTOM;	
57	if (is_up != 0)	
58	{	
59	if (is_down != 0)	
60	{	
61	if (is_down != 0)	
62	{	
63	if (is_down != 0)	
64	{	
65	if (is_down != 0)	
66	{	

Fringe Analysis	./fran/fringe_count.c	Page 3
132	int on_edge;	
133	float height_of_one_fringe;	
134	float height_of_scan_offset;	
135	float confidence;	
136	int length_of_edges_parallel_to_horizontal_scan_direction = 0;	
137	int edge_data;	
138		
139	height_of_one_fringe = 2.0 * PIF;	
140		
141	for(y = 0; y < ap.TILESIZE; y++)	
142	{	
143	offset = 0.0; nfringe_edges = 0; on_edge = 0;	
144		
145	for(x = 0; x < ap.TILESIZE; x++)	
146	{	
147	point_height = 0.0;	
148	theta = tile_phase[x+1][y+1];	
149	edge_data = (int) e[x][y];	
150		
151	if (edge_data == 0)	
152	{	
153	on_edge = 0;	
154	point_height = offset + theta;	
155		
156	else {	
157		
158	if (on_edge == 1)	
159	{	
160	length_of_edges_parallel_to_horizontal_scan_direction++;	
161	point_height = offset + theta;	
162		
163	else	
164	{	
165	is_up = edge_data & STEP_UP_EDGE_LEFT_TO_RIGHT;	
166	is_down = edge_data & STEP_DOWN_EDGE_LEFT_TO_RIGHT;	
167		
168	if (is_up != 0)	
169	{	
170	if (theta < 0.0) /* i.e the step hasn't happened yet */	
171	{	
172	point_height = offset + theta;	
173	offset -= height_of_one_fringe;	
174		
175	else /* the step has happened */	
176	{	
177	offset -= height_of_one_fringe;	
178	point_height = offset + theta;	
179		
180		
181	nfringe_edges++;	
182	on_edge = 1;	
183		
184	if (is_down != 0)	
185	{	
186	if (theta > 0.0) /* i.e the step hasn't happened yet */	
187	{	
188	point_height = offset + theta;	
189	offset += height_of_one_fringe;	
190		
191	else /* the step has happened */	
192	{	
193	offset += height_of_one_fringe;	
194	point_height = offset + theta;	
195		
196		
197		

Fringe Analysis	./fran/fringe_count.c	Page 4
198	nfringe_edges++; /* Yes : increment count of edges */	
199	on_edge = 1;	
200		
201		
202		
203		
204		
205	/* End Of Step Routine */	
206		
207	if (x == 0) start_of_scan_offset = point_height;	
208		
209	point_height -= start_of_scan_offset;	
210		
211	h[x][y] = point_height;	
212		
213	num_fringes_passed[y] = nfringe_edges;	
214		
215		
216	confidence = (float)	
217	{ ap.TILESIZE*ap.TILESIZE - length_of_edges_parallel_to_horizontal_scan_directio	
218	n };	
219		
220	return(confidence);	
221		
222	int* Allocate_Integer_Array(n)	
223	unsigned n;	
224	{	
225	int* p;	
226		
227		
228	p = (int*) Calloc(n , sizeof(int));	
229		
230	if (p == NULL)	
231	ray\n");	
232		
233	return(p);	
234		
235		
236	float* Allocate_Float_Array(n)	
237	unsigned n;	
238	{	
239	float* p;	
240		
241		
242	p = (float*) Calloc(n , sizeof(float));	
243		
244	if (p == NULL)	
245	fran_Error(fran_module, "Ran out of memory trying to allocate float array	
246	\n");	
247		
248	return(p);	

```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  *
5  */
6  typedef struct neighb {
7
8      struct xy_node* p;
9      unsigned long w;
10     float d;
11
12 } NEIGHB;
13
14 typedef struct xy_node
15 {
16     int x,y;
17
18     struct neighb n[4];
19
20     struct xy_node* pred;
21     struct xy_node* succ;
22
23 } XY_NODE;
24
25 typedef struct index
26 {
27     struct xy_node* p; long n; } INDEX;
28
29 typedef struct buffer
30 {
31     struct xy_node* memory;
32
33     int nel;
34     int max;
35
36     struct xy_node* head;
37     struct index cnode;
38     struct index lnode;
39
40 } BUFF;
41
42 void Init Graph();
43 void Reset Graph();
44 XY_NODE* Append Node();
45 void Move Node();
46 void Del Node();
47 XY_NODE* Set_Curr_Node();
48 long Curr_Node();
49 Last_Node();
50 extern BUFF Graph;
```

Fringe Analysis	./fran/graph.c	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	#include <stdio.h>	
6	#include <math.h>	
7	#include "franhader.h"	
8	#ifdef turbo	
9	#include <fcntl.h>	
10	#include <alloc.h>	
11	#endif	
12	#ifdef unixc	
13	#include <malloc.h>	
14	#include <urses.h>	
15	#endif	
16	#include "memory.h"	
17	#include "dynamicarray.h"	
18	#include "graph.h"	
19	#include "graph.h"	
20	#include "graph.h"	
21	BUFF Graph;	
22		
23	void Init Graph(nnode)	
24	unsigned nnode;	
25	{	
26	Graph.memory = (XY_NODE*)	
27	Malloc((unsigned) (sizeof(XY_NODE) * (nnode+1)));	
28		
29	if (Graph.memory != NULL)	
30	{	
31	Graph.nel = 0;	
32	Graph.nx = 0;	
33	Graph.ny = 0;	
34	Graph.head = (Graph.memory[inode]);	
35	Graph.head->succ = Graph.head;	
36	Graph.head->pred = Graph.head;	
37	Graph.inode.n = 0;	
38	Graph.inode.p = Graph.head;	
39	Graph.cnode = Graph.inode;	
40	}	
41		
42		
43	void Reset_Graph()	
44	{	
45	Graph.nel = 0;	
46	Graph.head->succ = Graph.head;	
47	Graph.head->pred = Graph.head;	
48	Graph.inode.n = 0;	
49	Graph.inode.p = Graph.head;	
50	Graph.cnode = Graph.inode;	
51	}	
52		
53	XY_NODE* Append_Node()	
54	{	
55	XY_NODE* node;	
56	XY_NODE* p;	
57	if (Graph.nel < Graph.max)	
58	node = (Graph.memory[Graph.nel++]);	
59	else {	
60	node = (Graph.memory[Graph.nel++]);	
61	else {	
62	p = Graph.inode.p;	
63	node->succ = p->succ;	
64	p->succ->pred = node;	
65	node->pred = p;	
66	}	

Fringe Analysis	./fran/graph.c	Page 2
67	p->succ	
68	= node;	
69	Graph.inode.p = Graph.head->pred;	
70	Graph.inode.n++;	
71	}	
72	Graph.cnode = Graph.inode;	
73	return(node);	
74	}	
75		
76	void Move_Node(p, node)	
77	XY_NODE* p, *node;	
78	{	
79	if ((node != NULL) && (p != NULL))	
80	{	
81	node->pred->succ = node->succ;	
82	node->succ->pred = node->pred;	
83	node->succ	
84	= p->succ;	
85	p->succ->pred = node;	
86	node->pred = p;	
87	p->succ	
88	= node;	
89	Graph.inode.p = Graph.head->pred;	
90	Graph.cnode = Graph.inode;	
91	}	
92	else {	
93	Fran_Error(fran_module, "Attempt to move NULL node!");	
94	}	
95	void Del_Node(p)	
96	XY_NODE* p;	
97	{	
98	if (p != NULL)	
99	{	
100	p->pred->succ = p->succ;	
101	p->succ->pred = p->pred;	
102	Graph.inode.p = Graph.head->pred;	
103	Graph.inode.n--;	
104	Graph.inode.p = Graph.inode;	
105	Graph.cnode = Graph.inode;	
106	}	
107	else {	
108	Fran_Error(fran_module, "Attempt to delete NULL node!");	
109	}	
110	XY_NODE* Set_Curr_Node(n)	
111	long n;	
112	{	
113	long nstep;	
114	long diff;	
115	XY_NODE* p = Graph.head;	
116	XY_NODE* n = Graph.inode.n;	
117	if (n > Graph.inode.n) n = Graph.inode.n;	
118	nstep = n;	
119	diff = n - Graph.cnode.n;	
120	if (abs((int) diff) < nstep) { p = Graph.cnode.p; nstep = diff; }	
121	diff = n - Graph.inode.n;	
122	if (abs((int) diff) < abs((int) nstep))	
123	{ p = Graph.inode.p; nstep = diff; }	
124	}	
125	if (nstep >= 0) { while(nstep--) p=p->succ; }	
126	else { while(nstep++) p=p->pred; }	
127	}	
128	Graph.cnode.n = n;	
129	Graph.cnode.p = p;	
130	}	
131		
132		

```
133     return( p );  
134 }  
135  
136 long Curr_Node()  
137 { return( Graph.cnode.n ); }  
138  
139 long Last_Node()  
140 { return( Graph.lnode.n ); }
```

Fringe Analysis	./fran/imageprepro.h	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*	
5	*/	
6	extern void Prepare_Images();	
7	extern void Add_Free_To_Fringe_Edge_Image();	

Fringe Analysis	./fran/imageprepro.c	Page 2
67	char string[80];	
68	unsigned width;	
69	unsigned height;	
70	unsigned char* raster;	
71	unsigned x;	
72	unsigned y;	
73	int grey_level;	
74	int i;	
75		
76	tif_in = TIFFOpen(input_fname, "r");	
77	if (tif_in == NULL) {	
78	sprintf(string,	
79	"Couldn't open tif image file %s\n", input_fname);	
80	Ffran_Error(fran_module, string);	
81	}	
82	height = (unsigned) tif_in->tif_dir.td_imagelength;	
83	width = (unsigned) tif_in->tif_dir.td_imagewidth;	
84	raster = (unsigned char*) Malloc(TIFFScanlinesize(tif_in));	
85	if (raster == NULL)	
86	Ffran_Error(fran_module, "Ran out of memory trying to allocate tif raster"	
87);	
88	*max = 0; *min = 65535;	
89	for(y = 0; y < height; y++)	
90	{	
91	Read_Scanline(tif_in, raster, y);	
92	if (tif_in->tif_dir.td_bitspersample == 8)	
93	{	
94	for(x = 0; x < width; x++)	
95	{	
96	if ((grey_level = (int) raster[x]) > *max) *max = grey_level;	
97	if ((grey_level < *min)) *min = grey_level;	
98	}	
99	}	
100	if (tif_in->tif_dir.td_bitspersample == 16)	
101	{	
102	for(x = 0; x < width; x++)	
103	{	
104	if ((grey_level = (int) raster[x]) > *max) *max = grey_level;	
105	if ((grey_level < *min)) *min = grey_level;	
106	}	
107	}	
108	if (tif_in->tif_dir.td_bitspersample == 16)	
109	{	
110	for(x = 0; x < width; x++)	
111	{	
112	if ((grey_level = (int) ((unsigned short*) raster)[x]) > *max)	
113	*max = grey_level;	
114	if ((grey_level < *min)) *min = grey_level;	
115	}	
116	}	
117	TIFFClose(tif_in);	
118	Free(raster);	
119	}	
120		
121	void Tif_Norm(input_fname, output_fname, min_of_all, max_of_all)	
122	char* input_fname;	
123	char* output_fname;	
124	int min_of_all, max_of_all;	
125	{	
126	TIFF* tif_in;	
127	char* tif_out;	
128	char* string[80];	
129	unsigned width;	
130	unsigned height;	
131	unsigned char* raster;	

Fringe Analysis	./fran/imageprepro.c	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	#include <stdio.h>	
6	#include <math.h>	
7	#include <assert.h>	
8	#include <fcntl.h>	
9	#include "franhdr.h"	
10	#define turboc	
11	#include <alloc.h>	
12	#include <dir.h>	
13	#include <io.h>	
14	#endif	
15	#ifdef unix	
16	#include <urses.h>	
17	#include <malloc.h>	
18	#endif	
19	#include "tiffio.h"	
20	#include "memory.h"	
21	#include "dynamicarray.h"	
22	#include "main.h"	
23	#include "computewrapped.h"	
24	#include "imageprepro.h"	
25		
26	int verbose;	
27		
28	void Delete(name)	
29	char* name;	
30	{	
31	if (verbose == 1) printf("Deleting %s\n", name);	
32	if (unlink(name) == -1) Ffran_Error(fran_module, "Delete failed!");	
33	}	
34		
35	void Rename(old, new)	
36	char* old;	
37	char* new;	
38	{	
39	char command[MAXPATH];	
40		
41	if (verbose == 1) printf("Renaming %s To %s\n", old, new);	
42		
43	#endif turboc	
44		
45	sprintf(command, "copy %s %s\n", old, new);	
46	system(command);	
47		
48	#endif	
49		
50	#endif unix	
51		
52	sprintf(command, "cp %s %s\n", old, new);	
53	system(command);	
54		
55	#endif	
56		
57	Delete(old);	
58	}	
59		
60	void Tif_Find_Min_Max(input_fname, min, max)	
61	char* input_fname;	
62	int* min;	
63	int* max;	
64	{	
65	TIFF* tif_in;	
66		

```

132 unsigned short* raster_out;
133 unsigned x;
134 unsigned y;
135 int grey_level;
136 int normalised = 0;
137 int i;
138
139 tif_in = TIFFOpen( input_frame, "r" );
140
141 if ( tif_in == NULL ) {
142     printf( string );
143     "Couldn't open tif image file %s\n", input_frame );
144     Fran_Error( fran_module, string );
145 }
146
147 height = (unsigned) tif_in->tif_dir.td_imagelength;
148 width = (unsigned) tif_in->tif_dir.td_imagewidth;
149
150 raster = (unsigned char*) Malloc( TIFFScanlineSize( tif_in ) );
151
152 if ( raster == NULL )
153     Fran_Error( fran_module, "Ran out of memory trying to allocate tif raster" );
154
155 tif_out = TIFFOpen( output_frame, "w" );
156
157 if ( tif_out == NULL ) {
158     printf( string );
159     "Couldn't open tif image file %s\n", output_frame );
160     Fran_Error( fran_module, string );
161 }
162
163 Setup_TIFF_Header( tif_out, tif_in, "\nNormalised\n", 16 );
164 raster_out = (unsigned short*) Malloc( TIFFScanlineSize( tif_out ) );
165
166 for( y = 0; y < height; y++ )
167     Read_Scanline( tif_in, raster, y );
168
169 /* The image is saved with a resolution of 16 bits per pixel */
170
171 if ( tif_in->tif_dir.td_bitspersample == 8 )
172     for( x = 0; x < width; x++ )
173         raster_out[x] = (unsigned short)
174             ((65535.0*((float) ((int) raster[x]) - min_of_all))
175              /((float) (max_of_all-min_of_all))+0.5);
176
177 if ( tif_in->tif_dir.td_bitspersample == 16 )
178     for( x = 0; x < width; x++ )
179         raster_out[x] = (unsigned short)
180             ((65535.0*((float) ((int) ((unsigned short*)raster)[x]) - min_of_all))
181              /((float) (max_of_all-min_of_all))+0.5);
182
183 TIFFWriteScanline( tif_out, (u_char *) raster_out, (u_int) y, (u_int) i);
184
185
186
187
188
189
190
191
192
193
194
195

```

Fringe Analysis ./fran/imageprepro.c Page 3

```

196
197 }
198
199 TIFFFlushData( tif_out );
200 TIFFClose( tif_out );
201 Free( raster_out );
202
203 TIFFClose( tif_in );
204 Free( raster );
205 }
206
207 int tif_Average( input_frame, output_frame )
208 char* input_frame;
209 char* output_frame;
210 {
211     TIFF* tif_in;
212     TIFF* tif_out;
213     char string[80];
214     unsigned width;
215     unsigned height;
216     unsigned x;
217     unsigned y;
218     unsigned yp;
219     int i;
220     int ifail = 0;
221     unsigned char* rasters[3];
222     unsigned short* raster_out;
223     unsigned char* temp;
224     int t;
225     unsigned sum_colm_1, sum_colm_2, sum_colm_3;
226     unsigned average;
227
228     tif_in = TIFFOpen( input_frame, "r" );
229     if ( tif_in == NULL ) {
230         printf( string );
231         "Couldn't open tif image file %s\n", input_frame );
232         Fran_Error( fran_module, string );
233     }
234
235     height = (unsigned) tif_in->tif_dir.td_imagelength;
236     width = (unsigned) tif_in->tif_dir.td_imagewidth;
237     for( i = 0; i < 3; i++ )
238         rasters[i] = (unsigned char*) Malloc( TIFFScanlineSize( tif_in ) );
239     if ( rasters[i] == NULL )
240         Fran_Error( fran_module, "Ran out of memory trying to allocate tif raster" );
241
242     if ( (height > 2) && (width > 2) ) {
243         tif_out = TIFFOpen( output_frame, "w" );
244         if ( tif_out == NULL ) {
245             printf( string );
246             "Couldn't open tif image file %s\n", output_frame );
247             Fran_Error( fran_module, string );
248         }
249     }
250 }
251
252
253
254
255
256
257
258
259
260

```

Fringe Analysis ./fran/imageprepro.c Page 4

Fringe Analysis	./fran/imageprepro.c	Page 6
326	raster_out[x] = (unsigned short) average;	
327	}	
328	TIFFWriteScanline(tif_out, (u_char *) raster_out,	
329	(u_int) (yp-2), (u_int) 0);	
330	r = 0;	
331	temp	
332	= rasters[0];	
333	rasters[0] = rasters[1]; r++;	
334	rasters[1] = rasters[2]; r++;	
335	rasters[2] = temp;	
336	}	
337	if (tif_in->tif_dir.td_bitspersample == 16)	
338	{	
339	raster_out[0] = ((unsigned short*) rasters[1])[0];	
340	raster_out[width-1] = ((unsigned short*) rasters[1])[width-1];	
341	for(x = 1; x < (width-1); x++)	
342	{	
343	if { x == 1 }	
344	{	
345	sum_colm_1 = (unsigned) (((unsigned short*) rasters[0])[x-	
346	((unsigned short*) rasters[1])[x-	
347	((unsigned short*) rasters[2])[x-	
348	sum_colm_2 = (unsigned) (((unsigned short*) rasters[0])[x	
349	((unsigned short*) rasters[1])[x	
350	((unsigned short*) rasters[2])[x	
351	sum_colm_3 = (unsigned) (((unsigned short*) rasters[0])[x+	
352	((unsigned short*) rasters[1])[x+	
353	((unsigned short*) rasters[2])[x+	
354	}	
355	}	
356	sum_colm_1 = sum_colm_2;	
357	sum_colm_2 = sum_colm_3;	
358	sum_colm_3 = (unsigned) (((unsigned short*) rasters[0])[x+	
359	((unsigned short*) rasters[1])[x+	
360	((unsigned short*) rasters[2])[x+	
361	}	
362	}	
363	}	
364	average = (sum_colm_1+sum_colm_2+sum_colm_3) / 9;	
365	raster_out[x] = (unsigned short) average;	
366	}	
367	TIFFWriteScanline(tif_out, (u_char *) raster_out,	
368	(u_int) (yp-2), (u_int) 0);	
369	}	
370	}	
371	}	
372	}	
373	}	
374	}	
375	}	
376	}	
377	}	
378	}	
379	}	

Fringe Analysis	./fran/imageprepro.c	Page 5
261	}	
262	Setup_TIFF_Header(tif_out, tif_in,	
263	"\nLow Pass Filtered by 3 x 3 Averaging Filter\n", 16);	
264	raster_out = (unsigned short*) Malloc(TIFFScanlineSize(tif_out));	
265	if (raster_out == NULL)	
266	{	
267	Ffran_Error(fran_module, "ran out of memory trying to allocate tif raster"	
268);	
269	yp = 0; r = 0;	
270	Read_Scanline(tif_in, rasters[r], yp);	
271	if (tif_in->tif_dir.td_bitspersample == 8)	
272	{	
273	for(x = 0; x < width; x++)	
274	{	
275	raster_out[x] = ((unsigned short) rasters[r][x]) << 8;	
276	}	
277	}	
278	if (tif_in->tif_dir.td_bitspersample == 16)	
279	{	
280	for(x = 0; x < width; x++)	
281	{	
282	raster_out[x] = ((unsigned short*) rasters[r])[x];	
283	}	
284	}	
285	TIFFWriteScanline(tif_out, (u_char *) raster_out, (u_int) yp, (u_int) 0);	
286	yp++; r++;	
287	while(yp < height)	
288	{	
289	while(r < 3) Read_Scanline(tif_in, rasters[r++], yp++);	
290	}	
291	if (tif_in->tif_dir.td_bitspersample == 8)	
292	{	
293	raster_out[0] = ((unsigned short) rasters[1][0]) << 8;	
294	raster_out[width-1] = ((unsigned short) rasters[1][width-1]) << 8;	
295	for(x = 1; x < (width-1); x++)	
296	{	
297	if { x == 1 }	
298	{	
299	sum_colm_1 = ((unsigned) rasters[0][x-1] +	
300	(unsigned) rasters[1][x-1] +	
301	(unsigned) rasters[2][x-1]);	
302	sum_colm_2 = ((unsigned) rasters[0][x] +	
303	(unsigned) rasters[1][x] +	
304	(unsigned) rasters[2][x]);	
305	sum_colm_3 = ((unsigned) rasters[0][x+1] +	
306	(unsigned) rasters[1][x+1] +	
307	(unsigned) rasters[2][x+1]);	
308	}	
309	}	
310	sum_colm_1 = sum_colm_2;	
311	sum_colm_2 = sum_colm_3;	
312	sum_colm_3 = ((unsigned) rasters[0][x+1] +	
313	(unsigned) rasters[1][x+1] +	
314	(unsigned) rasters[2][x+1]);	
315	}	
316	}	
317	}	
318	}	
319	}	
320	}	
321	}	
322	}	
323	}	
324	}	
325	}	
326	}	
327	}	
328	}	
329	}	
330	}	
331	}	
332	}	
333	}	
334	}	
335	}	
336	}	
337	}	
338	}	
339	}	
340	}	
341	}	
342	}	
343	}	
344	}	
345	}	
346	}	
347	}	
348	}	
349	}	
350	}	
351	}	
352	}	
353	}	
354	}	
355	}	
356	}	
357	}	
358	}	
359	}	
360	}	
361	}	
362	}	
363	}	
364	}	
365	}	
366	}	
367	}	
368	}	
369	}	
370	}	
371	}	
372	}	
373	}	
374	}	
375	}	
376	}	
377	}	
378	}	
379	}	
380	}	
381	}	
382	}	
383	}	
384	}	
385	}	
386	}	
387	}	
388	}	
389	}	
390	}	
391	}	
392	}	
393	}	
394	}	
395	}	
396	}	
397	}	
398	}	
399	}	
400	}	
401	}	
402	}	
403	}	
404	}	
405	}	
406	}	
407	}	
408	}	
409	}	
410	}	
411	}	
412	}	
413	}	
414	}	
415	}	
416	}	
417	}	
418	}	
419	}	
420	}	
421	}	
422	}	
423	}	
424	}	
425	}	
426	}	
427	}	
428	}	
429	}	
430	}	
431	}	
432	}	
433	}	
434	}	
435	}	
436	}	
437	}	
438	}	
439	}	
440	}	
441	}	
442	}	
443	}	
444	}	
445	}	
446	}	
447	}	
448	}	
449	}	
450	}	
451	}	
452	}	
453	}	
454	}	
455	}	
456	}	
457	}	
458	}	
459	}	
460	}	
461	}	
462	}	
463	}	
464	}	
465	}	
466	}	
467	}	
468	}	
469	}	
470	}	
471	}	
472	}	
473	}	
474	}	
475	}	
476	}	
477	}	
478	}	
479	}	
480	}	
481	}	
482	}	
483	}	
484	}	
485	}	
486	}	
487	}	
488	}	
489	}	
490	}	
491	}	
492	}	
493	}	
494	}	
495	}	
496	}	
497	}	
498	}	
499	}	
500	}	
501	}	
502	}	
503	}	
504	}	
505	}	
506	}	
507	}	
508	}	
509	}	
510	}	
511	}	
512	}	
513	}	
514	}	
515	}	
516	}	
517	}	
518	}	
519	}	
520	}	
521	}	
522	}	
523	}	
524	}	
525	}	
526	}	
527	}	
528	}	
529	}	
530	}	
531	}	
532	}	
533	}	
534	}	
535	}	
536	}	
537	}	
538	}	
539	}	
540	}	
541	}	
542	}	
543	}	
544	}	
545	}	
546	}	
547	}	
548	}	
549	}	
550	}	
551	}	
552	}	
553	}	
554	}	
555	}	
556	}	
557	}	
558	}	
559	}	
560	}	
561	}	
562	}	
563	}	
564	}	
565	}	
566	}	
567	}	
568	}	
569	}	
570	}	
571	}	
572	}	
573	}	
574	}	
575	}	
576	}	
577	}	
578	}	
579	}	
580	}	
581	}	
582	}	
583	}	
584	}	
585	}	
586	}	
587	}	
588	}	
589	}	
590	}	
591	}	
592	}	
593	}	
594	}	
595	}	
596	}	
597	}	
598	}	
599	}	
600	}	
601	}	
602	}	
603	}	
604	}	
605	}	
606	}	
607	}	
608	}	
609	}	
610	}	
611	}	
612	}	
613	}	
614	}	
615	}	
616	}	
617	}	
618	}	
619	}	
620	}	
621	}	
622	}	
623	}	
624	}	
625	}	
626	}	
627	}	
628	}	
629	}	
630	}	
631	}	
632	}	
633	}	
634	}	
635	}	
636	}	
637	}	
638	}	
639	}	
640	}	
641	}	
642	}	

Fringe Analysis	./fran/imageprepro.c	Page 7
380	z = 0;	
381	temp	
382	rasters[0] = rasters[0];	
383	rasters[0] = rasters[1]; i++;	
384	rasters[1] = rasters[2]; i++;	
385	rasters[2] = temp;	
386		
387		
388		
389		
390		
391		
392	if (tif_in->tif_dir.td.bitspersample == 8)	
393	{	
394	for(x = 0; x < width; x++) raster_out[x] = ((unsigned short) rasters[i]	
395	[x]) << 8;	
396	TiffWriteScanline(tif_out, (u_char *) raster_out, (u_int) (height-1), (u	
397	_int) 0);	
398		
399	if (tif_in->tif_dir.td.bitspersample == 16)	
400	{	
401	for(x = 0; x < width; x++) raster_out[x] = ((unsigned short*) rasters[i]	
402)[x];	
403	TiffWriteScanline(tif_out, (u_char *) raster_out, (u_int) (height-1), (u	
404	_int) 0);	
405		
406	TIFFFlushData(tif_out);	
407	TIFFClose(tif_out);	
408		
409	} else fail = 1;	
410	TIFFClose(tif_in);	
411		
412	for(i = 0; i < 3; i++)	
413	Free(rasters[i]);	
414		
415	Free(raster_out);	
416		
417	return(fail);	
418		
419		
420	static int intcompare(i,j)	
421	int *i, *j;	
422	{	
423	return(*i - *j);	
424	}	
425		
426	int TifMedian(input_frame, output_frame)	
427	char* input_fname;	
428	char* output_fname;	
429	{	
430	TIFF* tif_in;	
431	TIFF* tif_out;	
432	char string[80];	
433	unsigned width;	
434	unsigned height;	
435	unsigned x;	
436	unsigned y;	
437	unsigned yp;	
438	int i, j, k;	
439	int fail = 0;	
440	unsigned char* rasters[3];	
441	unsigned short* raster_out;	

Fringe Analysis	./fran/imageprepro.c	Page 8
442	unsigned char* temp;	
443	int i;	
444	int median;	
445	int list_to_sort[9];	
446		
447	tif_in = TiffOpen(input_fname, "r");	
448		
449	if (tif_in == NULL) {	
450	fprintf(string,	
451	"Couldn't open tif image file %s\n", input_fname);	
452	Fran_Error(fran_module, string);	
453		
454		
455		
456		
457	if (verbose == 1) printf("Averaging Source: %s To Dest: %s\n", input_fname, out	
458	put_fname);	
459		
460	height = (unsigned) tif_in->tif_dir.td.imagelength;	
461	width = (unsigned) tif_in->tif_dir.td.imagewidth;	
462		
463	for(i = 0; i < 3; i++)	
464	{	
465	rasters[i] = (unsigned char*) Malloc(TIFFScanlineSize(tif_in));	
466	if (rasters[i] == NULL)	
467	Fran_Error(fran_module, "Ran out of memory trying to allocate tif ras	
468	ter");	
469		
470		
471		
472	if ((height > 2) && (width > 2)) {	
473		
474	tif_out = TiffOpen(output_fname, "w");	
475		
476	if (tif_out == NULL) {	
477	fprintf(string,	
478	"Couldn't open tif image file %s\n", output_fname);	
479	Fran_Error(fran_module, string);	
480		
481		
482		
483		
484		
485	Setup_TIFF_Header(tif_out, tif_in,	
486	"\nLow Pass Filtered by 3 x 3 Median Filter\n", 16);	
487		
488	raster_out = (unsigned short*) Malloc(TIFFScanlineSize(tif_out));	
489		
490	if (raster_out == NULL)	
491	Fran_Error(fran_module, "Ran out of memory trying to allocate tif raster"	
492);	
493	yp = 0; i = 0;	
494	Read_Scanline(tif_in, rasters[i], yp);	
495		
496	if (tif_in->tif_dir.td.bitspersample == 8)	
497	{	
498	for(x = 0; x < width; x++)	
499	raster_out[x] = ((unsigned short) rasters[i][x]) << 8;	
500		
501		
502	if (tif_in->tif_dir.td.bitspersample == 16)	
503		
504		

Fringe Analysis	./fran/imageprepro.c	Page 11
635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699	<pre> k = 0; while(list_to_sort[k] <= list_to_sort[8]) k++; if (k < 8) { bcopy(&list_to_sort[k], &list_to_sort[k+1], ((8-k-1)*(int)sizeof(int))); list_to_sort[k] = list_to_sort[k+1]; (int) ((unsigned short*) rasters[i])[x+1]; break; } /* median is greater than or equal to 4 elements, must be 5th in rank, but index starts at 0 and so median has index of 4 */ raster_out[x] = (unsigned short) list_to_sort[4]; } TIFFWriteScanline(tif_out, (u_char *) raster_out, (u_int) (yp-2), (u_int) 0); z = 0; temp = rasters[0]; rasters[0] = rasters[1]; r++; rasters[1] = rasters[2]; i++; rasters[2] = temp; } for(x = 0; x < width; x++) raster_out[x] = ((unsigned short*) rasters[i])[x]; TIFFWriteScanline(tif_out, (u_char *) raster_out, (u_int) (height-1), (u_int) 0); } TIFFFlushData(tif_out); TIFFClose(tif_out); } else fail = 1; TIFFClose(tif_in); for(i = 0; i < 3; i++) Free(rasters[i]); Free(raster_out); return(fail); } int tree_buffer_sort(a, b) int* a; int* b; { if (*(a+1) < *(b+1)) return(-1); else {</pre>	

Fringe Analysis	./fran/imageprepro.c	Page 12
700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764	<pre> if (*(a+1) > *(b+1)) return(1); else { if (*a < *b) return(-1); else { if (*a > *b) return(1); else return(0); } } } void Add_Tree_To_Fringe_Edge_Image(tree_frame, tif_frame) char* tree_frame; char* tif_frame; { TIFF* tif_in; TIFF* tif_out; char string[80]; unsigned width; unsigned height; unsigned char* raster; unsigned x; unsigned y; char output_filename[80]; FILE* tree_in; FILE* tree_out; long tree_length; int* tree_buffer; int ydone; unsigned i; unsigned x1,y1,x2,y2; unsigned ix; unsigned number_of_ints; strcpy(output_filename, "XXXXXX.tmp"); tif_in = TIFFOpen(tif_frame, "r"); if (tif_in == NULL) { sprintf(string, "Couldn't open tif image file %s\n", tif_frame); Fran_Error(fran_module, string); } height = (unsigned) tif_in->tif_dir.td_imagelength; width = (unsigned) tif_in->tif_dir.td_imagewidth; raster = (unsigned char*) Malloc(TIFFScanlinesize(tif_in)); if (raster == NULL) Fran_Error(fran_module, "Ran out of memory trying to allocate tif raster="); tif_out = TIFFOpen(output_filename, "w"); if (tif_out == NULL) { sprintf(string, "Couldn't open tif image file %s\n", output_filename); Fran_Error(fran_module, string); }</pre>	

Fringe Analysis	./fran/imageprepro.c	Page 14
829	ydone[i] = 0;	
830		
831	nnl = 0;	
832		
833	x1 = ((unsigned) tree_buffer[nnl]) * ap.XSTEP + (ap.XSTEP/2);	
834	y1 = ((unsigned) tree_buffer[nnl+1]) * ap.XSTEP + (ap.XSTEP/2);	
835	x2 = ((unsigned) tree_buffer[nnl+2]) * ap.XSTEP + (ap.XSTEP/2);	
836	y2 = ((unsigned) tree_buffer[nnl+3]) * ap.XSTEP + (ap.XSTEP/2);	
837		
838	nnl += 4;	
839		
840	assert(x1 >= 0 && x1 < ap.PIXX);	
841	assert(y1 >= 0 && y1 < ap.PIXY);	
842		
843	assert(x2 >= 0 && x2 < ap.PIXX);	
844	assert(y2 >= 0 && y2 < ap.PIXY);	
845		
846	for(y = 0; y < height; y++)	
847	{	
848	Read_Scanline(tif_in, raster, y);	
849	while(y1 == y)	
850	{	
851	if (y1 == y2)	
852	{	
853	for(i = x1; i <= x2; i++) raster[i] = GREYSCALE_TREEMARK;	
854	}	
855	else	
856	{	
857	raster[x1] = GREYSCALE_TREEMARK;	
858	ix = ((x1-ap.XSTEP/2)/ap.XSTEP);	
859	assert((ix >= 0) && (ix < ap.NTILESX));	
860	ydone[ix] = y2;	
861	}	
862	}	
863	if (nnl >= number_of_ints) { y1 = height; break; }	
864		
865	x1 = ((unsigned) tree_buffer[nnl]) * ap.XSTEP + (ap.XSTEP/2);	
866	y1 = ((unsigned) tree_buffer[nnl+1]) * ap.XSTEP + (ap.XSTEP/2);	
867	x2 = ((unsigned) tree_buffer[nnl+2]) * ap.XSTEP + (ap.XSTEP/2);	
868	y2 = ((unsigned) tree_buffer[nnl+3]) * ap.XSTEP + (ap.XSTEP/2);	
869		
870	/*	
871	assert(x1 >= 0 && x1 < ap.PIXX);	
872	assert(y1 >= 0 && y1 < ap.PIXY);	
873		
874	assert(x2 >= 0 && x2 < ap.PIXX);	
875	assert(y2 >= 0 && y2 < ap.PIXY);	
876	nnl += 4;	
877	}	
878		
879	for(i = 0; i < ap.NTILESX; i++)	
880	{	
881	if (ydone[i] != 0)	
882	{	
883	if (y <= ydone[i])	
884	{	
885	ix = (i*ap.XSTEP)+(ap.XSTEP/2);	
886	assert((ix >= 0) && (ix < ap.PIXX));	
887	raster[ix] = GREYSCALE_TREEMARK;	
888	}	
889	}	
890	else ydone[i] = 0;	
891	}	
892		
893		
894		

Fringe Analysis	./fran/imageprepro.c	Page 13
765	}	
766	Setup_TIFF_Header(tif_out, tif_in, "\nfile to Tile Unwrap Tree\n", 8);	
767		
768	#ifdef turboc	
769	tree_in = fopen(tree_fname, "rb");	
770		
771	#endif	
772		
773	#ifdef unixc	
774	tree_in = fopen(tree_fname, "r");	
775		
776	#endif	
777		
778	if (tree_in == NULL) Frn_Error(frn_module, "Couldn't open tree file!");	
779		
780	fseek(tree_in, 0L, SEEK_END);	
781	tree_length = ftell(tree_in);	
782		
783	rewind(tree_in);	
784		
785	tree_buffer = (int*) Malloc((unsigned) tree_length);	
786		
787	if (tree_buffer == NULL) Frn_Error(frn_module, "Ran Out of Memory Trying	
788	to Allocate Tree Buffer");	
789		
790	if (fread(tree_buffer, sizeof(char), (unsigned) tree_length, tree_in) == 0)	
791	Frn_Error(frn_module, "File Read of Unwrap Tree");	
792	fclose(tree_in);	
793		
794	number_of_ints = (unsigned) (tree_length / ((long) sizeof(int)));	
795	qsort(tree_buffer, (number_of_ints / 4),	
796	(unsigned) (4*sizeof(int)),	
797	tree_buffer_sort);	
798		
799	#ifdef turboc	
800	tree_out = fopen(tree_fname, "wb");	
801		
802	#endif	
803		
804	#ifdef unixc	
805	tree_out = fopen(tree_fname, "w");	
806		
807	#endif	
808		
809	if (tree_out == NULL)	
810	Frn_Error(frn_module, "Couldn't open tree file!");	
811		
812	if (fwrite(tree_buffer, sizeof(char), (unsigned) tree_length, tree_out) == 0)	
813	Frn_Error(frn_module, "File Write of Unwrap Tree");	
814		
815	fclose(tree_out);	
816		
817	ydone = (int*) Malloc((unsigned) (ap.NTILESX * sizeof(int)));	
818		
819	if (ydone == NULL) Frn_Error(frn_module, "Ran Out of Memory Trying to Allo	
820	cate Buffer");	
821		
822	for(i = 0; i < ap.NTILESX; i++)	
823		
824		
825		
826		
827		
828		

```

895     }
896     TiffWriteScanline( tif_out, (u_char *) raster, (u_int) y, (u_int) 0);
897 }
898
899 Free( tree_buffer );
900 Free( ydone );
901
902 TIFFFlushData( tif_out );
903 TIFFClose( tif_out );
904
905 TIFFFlushData( tif_in );
906
907 TIFFClose( tif_in );
908
909 Free( raster );
910
911 Delete( tif_fname );
912
913 Rename( output_fname, tif_fname );
914
915 void Prepare_Images( average, median, normalise, nimages )
916 int average;
917 int median;
918 int normalise;
919 int nimages;
920 {
921     int i, j, k;
922     int c;
923     char input_fname[MAXPATH];
924     char output_fname[MAXPATH];
925     char temp_fname[MAXPATH];
926     int iterations;
927     char* src;
928     char* des;
929     char* tmp;
930     char* temp;
931     int result;
932     int iterations;
933     int min[ NIMAGES ];
934     int max[ NIMAGES ];
935     int min_of_all;
936     int max_of_all;
937     unsigned short norm(256);
938
939     if ( average != 0 || median != 0 || normalise != 0 )
940     {
941         strcpy( temp_fname, "XXXXXX.tmp" );
942         verbose = 0;
943         for( j = 0; j < nimages; j++ )
944         {
945             strcpy( input_fname, sp.file_names[j] );
946             k = 0; while( ( c = input_fname[k] ) != ',' ) k++;
947             input_fname[k] = '\0';
948             sprintf( output_fname, "%s.prp", input_fname );
949             if ( c == ',' ) input_fname[k] = c;
950             iterations = 0;
951         }
952     }
953 }

```

```

961     if ( average > 0 ) iterations = average;
962     else if ( median > 0 ) iterations = median;
963     if ( iterations > 0 )
964     {
965         src = input_fname;
966         des = output_fname;
967         for( i = 0; i < iterations; i++ )
968         {
969             if ( average > 0 ) {
970                 if ( ( result = Tif_Average( src, des ) ) == 1 )
971                 {
972                     prn( "%s Image Too Small To Filter!\n", src );
973                     break;
974                 }
975             }
976             else {
977                 if ( ( result = Tif_Median( src, des ) ) == 1 )
978                 {
979                     prn( "%s Image Too Small To Filter!\n", src );
980                     break;
981                 }
982             }
983             if ( i == 0 ) src = temp_fname;
984             tmp = src;
985             src = des;
986             des = tmp;
987         }
988     }
989     if ( ( iterations & 2 ) == 0 )
990     {
991         Rename( temp_fname, output_fname );
992         Delete( temp_fname );
993     }
994     if ( ( iterations > 1 ) && ( iterations & 2 ) == 1 )
995     {
996         if ( normalise == 1 )
997         {
998             if ( iterations > 0 )
999             {
1000                 Tif_Find_Min_Max( output_fname, &min[j], &max[j] );
1001                 Tif_Find_Min_Max( input_fname, &min[j], &max[j] );
1002                 min_of_all = 65535;
1003                 max_of_all = 0;
1004                 for( j = 0; j < nimages; j++ )
1005                 {
1006                     if ( min_of_all > min[j] ) min_of_all = min[j];
1007                 }
1008             }
1009         }
1010     }
1011 }

```

```
1027     if ( max_of_all < max[j] ) max_of_all = max[j];
1028 }
1029
1030 prn("Intensity Range of Un-normalised Images (%d -> %d)\n",
1031     min_of_all, max_of_all );
1032
1033 for( i = 0; i < 256; i++ )
1034     norm[i] = (unsigned short)
1035         ((65535.0*((float) (i - min_of_all))/((float)
1036             (max_of_all-min_of_all)))+0.5);
1037
1038 for( j = 0; j < nimages; j++ )
1039 {
1040     strcpy( input_fname, ap.file_names[j] );
1041     k = 0; while( ( ( c = input_fname[k] ) != ',' ) && ( c != '\0' ) ) k++;
1042     input_fname[k] = '\0';
1043     sprintf( output_fname, "%s.prp", input_fname );
1044     if ( c == ',' ) input_fname[k] = c;
1045     strcpy( ap.file_names[j], output_fname );
1046     if ( iterations > 0 )
1047     {
1048         Tif Norm( output_fname, temp_fname, min_of_all, max_of_all );
1049         Rename( temp_fname, output_fname );
1050     }
1051     else Tif Norm( input_fname, output_fname, min_of_all, max_of_all );
1052 }
1053
1054 }
1055
1056 }
1057
1058 }
1059
1060 }
```

Fringe Analysis	./fran/main.h	Page 1
<pre> 1 extern TILES ta; 2 extern PHASE phase; 3 extern int leftx; 4 extern int lefty; 5 extern int topy; 6 extern char file_names[3][80]; 7 extern void fran_error(); 8 extern int dump_its_begin_scan; 9 extern int dump_its_end_scan; 10 extern int window; 11 extern IMAGE edge_image; 12 extern int save edge_detect; 13 extern char* fran_module; </pre>		

Fringe Analysis	./fran/main.c	Page 2
<pre> 67 #ifdef turboc 68 c[0] = 218; 69 c[1] = 181; 70 c[2] = 172; 71 c[3] = 182; 72 c[4] = 182; 73 c[5] = 179; 74 c[6] = 196; 75 c[7] = 179; 76 #endif 77 78 #ifdef unixc 79 c[0] = '/'; 80 c[1] = '\\'; 81 c[2] = '\\'; 82 c[3] = '/'; 83 c[4] = '-'; 84 c[5] = '+'; 85 c[6] = '+'; 86 c[7] = '+'; 87 #endif 88 89 printat(1, 1); prn("%c", c[0]); 90 printat(1, 79); prn("%c", c[1]); 91 printat(24, 1); prn("%c", c[2]); 92 printat(24, 79); prn("%c", c[3]); 93 94 for(y = 2; y < 24; y++) 95 { 96 printat(y, 1); prn("%c", c[5]); 97 printat(y, 79); prn("%c", c[7]); 98 } 99 100 for(x = 2; x < 79; x++) 101 { 102 printat(1, x); prn("%c", c[4]); 103 printat(24, x); prn("%c", c[6]); 104 } 105 106 #ifdef unixc 107 refresh(); 108 #endif 109 110 } 111 112 void Exit(n) 113 { 114 int n; 115 } 116 117 #ifdef unixc 118 refresh(); 119 #endif 120 121 exit(n); 122 } 123 124 int MarkDeadPixels(tile_phase, max_dead_pixel_count) 125 PHASE tile_phase; 126 unsigned max_dead_pixel_count; 127 { 128 int x,y; 129 int px,py; 130 int minx,miny; 131 int maxx,maxy; 132 </pre>		

Fringe Analysis	./fran/main.c	Page 1
<pre> 1 /* 2 * Copyright (c) 1991 by Tom Judge. 3 * All rights reserved. 4 */ 5 6 #include <stdio.h> 7 #include <values.h> 8 #include <math.h> 9 #include <string.h> 10 #include <time.h> 11 #include <signal.h> 12 #include <assert.h> 13 #include "franhdr.h" 14 #include "franhdr.h" 15 #ifdef turboc 16 #include <fcntl.h> 17 #include <alloc.h> 18 #include <dir.h> 19 #endif 20 #ifdef unixc 21 #include <varargs.h> 22 #include <unistd.h> 23 #include <malloc.h> 24 #endif 25 #include "tiffio.h" 26 #include "memory.h" 27 #include "correctoffsets.h" 28 #include "fringecount.h" 29 #include "dynamicarray.h" 30 #include "polysmooth.h" 31 #include "imageprepro.h" 32 #include "computewrapped.h" 33 #include "tileelements.h" 34 #include "tfileutils.h" 35 #include "main.h" 36 37 ANALYSIS_PARAMETERS ap; 38 ANALYSIS_PARAMETERS last_ap; 39 40 #ifdef turboc 41 #define TICKS ((double) CLK_TCK) 42 #endif 43 #ifdef unixc 44 #define TICKS ((double) 1000000.0) 45 #endif 46 47 int windowing; 48 49 TILES ta; /* ta[i][j] is the tile array */ 50 PHASE phase; /* phase[i][j] is the wrapped phase map */ 51 52 int leftx,topy; 53 54 unsigned int dead_pixel_count; 55 unsigned int fringe_edge_set_pixel_count; 56 unsigned int fringe_edge_end_point_count; 57 58 BYTES edge_image; 59 TIFF* edge_out; 60 char* fran_module; 61 62 void Draw_Border() 63 { 64 int x,y; 65 int c[8]; 66 </pre>		


```

133 int ok = 1;
134 int n;
135 int p_leftx;
136 int p_tcopy;
137
138 dead_pixel_count = 0;
139
140 p_leftx = leftx-1;
141 p_tcopy = 0; /* This would be (copy-1), however the phase array is
142                computed in chunks so that the y coordinate of the first
143                pixel in any tile is always 1 in the array phase[][],
144                the tile phase[][] array needs an extra pixel around the
145                edge of the tile for edge detection to be performed on a
146                tile by tile basis
147                */
148
149 minx = 0; miny = 0; /* define limits in tile phase[][] array */
150 maxx = minx+ap.TILESIZE*2; maxy = miny+ap.TILESIZE*2;
151
152 for( y = miny; y < maxy; y++ )
153     for( x = minx; x < maxx; x++ )
154     {
155         px = p_leftx + x; py = p_tcopy + y;
156
157         if ( ( ( px < 0 ) || ( px >= ap.PIXX ) ) ) {
158             if ( phase[px][py] > LIMIT_HEIGHT )
159                 tile_phase[x][y] = DEAD_PIXEL;
160             dead_pixel_count++;
161             if ( ( y < ap.YSTEP ) && ( ap.save_edge_detect == 1 ) )
162                 if ( ( (x-leftx) < ap.PIXX )
163                     && edge_image[x][leftx][y] = GREYSCALE_GREY;
164         }
165     }
166
167 if ( dead_pixel_count > max_dead_pixel_count ) ok = 0;
168
169 return( ok );
170
171 void GenerateTilePhaseArray( tile_phase )
172 PHASE tile_phase;
173 {
174     int i,j;
175     int x,y;
176     int xi,yi;
177     int px,py;
178     int minx,miny;
179     int maxx,maxy;
180     int ok = 1;
181     int n;
182     int p_leftx;
183     int p_tcopy;
184
185     p_leftx = leftx-1;
186     p_tcopy = 0; /* This would be (copy-1), however the phase array is
187                  computed in chunks so that the y coordinate of the first
188                  pixel in any tile is always 1 in the array phase[][],
189                  */

```

```

199     the tile_phase[][] array needs an extra pixel around the
200     edge of the tile for edge detection to be performed on a
201     tile by tile basis
202     */
203
204 minx = 0; miny = 0; /* define limits in tile_phase[][] array */
205 maxx = minx+ap.TILESIZE*2; maxy = miny+ap.TILESIZE*2;
206
207 for( y = miny; y < maxy; y++ )
208     for( x = minx; x < maxx; x++ )
209     {
210         px = p_leftx + x; py = p_tcopy + y;
211
212         if ( ( px < 0 ) || ( px >= ap.PIXX ) )
213             /* don't need to check y */
214             tile_phase[x][y] = 0.0;
215         else tile_phase[x][y] = phase[px][py];
216
217         if ( tile_phase[x][y] > LIMIT_HEIGHT ) tile_phase[x][y] = 0.0;
218     }
219
220 void Dcr( string )
221 char* string;
222 {
223     char* p;
224     if ( string != NULL )
225     {
226         p = string;
227         while ( *p != '\0' && *p != '\n' ) p++;
228         if ( *p == '\n' ) *p = '\0';
229     }
230
231 int SobelEdgeDetectWithHysteresisThresholding
232 ( tile_phase, sobel_mode, low_threshold, high_threshold, dir )
233 PHASE tile_phase;
234 int sobel_mode;
235 float low_threshold;
236 float high_threshold;
237 BYTES dir;
238 {
239     int x,y;
240     int xi,yi;
241     int minx,miny;
242     int maxx,maxy;
243     float delta1;
244     float delta2;
245     int edges_found = 0;
246     int low_threshold_edge_found;
247     int no_neighbours_left_count;
248     int above_high;
249     int box_side_touch_count;
250     int box_side_touch_flags[4];
251     int i;
252     fringe_edge_end_point_count = 0;
253
254     /* sobel_mode of 1 indicates hysteresis thresholding using
255        the high threshold to pull out confident edges and the
256        low threshold to pull out edges adjoining the confident
257        detections.
258    */

```

Fringe Analysis	./fran/main.c	Page 6
<pre> 329 330 if (delta2 > 0.0) 331 dir[x1][y1] = 332 (unsigned char) 333 (((int) dir[x1][y1]) STEP_DOWN_EDGE_TOP_TO_BOTTOM); 334 335 if (delta2 < 0.0) 336 dir[x1][y1] = 337 (unsigned char) 338 (((int) dir[x1][y1]) STEP_UP_EDGE_TOP_TO_BOTTOM); 339 340 } 341 342 if (sobel_mode == 1) { 343 do { 344 low_threshold_edge_found = 0; 345 for (y = miny; y < maxy; y++) 346 for (x = minx; x < maxx; x++) 347 { 348 x1 = x - minx; y1 = y - miny; 349 if (350 { ((int) dir[x1][y1]) & (ABOVE_HIGH_THRESHOLD) != 0 } && 351 { ((int) dir[x1][y1]) & (NO_NEIGHBOURS_LEFT) == 0 } 352) 353 no_neighbours_left_count = 0; 354 } 355 for (y2 = y1 - 1; y2 <= y1 + 1; y2++) 356 for (x2 = x1 - 1; x2 <= x1 + 1; x2++) 357 { 358 if (x2 >= 0 && (x2 < (maxx - minx)) && 359 { (x2+leftx) < ap.PIXX } && 360 y2 >= 0 && (y2 < (maxy - miny))) 361 { 362 if ((x1 != x2) (y1 != y2)) && 363 { ((int) dir[x2][y2]) != 0 } && 364 { ((int) dir[x2][y2]) & 365 (ABOVE_HIGH_THRESHOLD) == 0 } 366) 367 { 368 dir[x2][y2] = 369 (unsigned char) 370 { ((int) dir[x2][y2]) 371 ABOVE_HIGH_THRESHOLD }; 372 } 373 } 374 } 375 } 376 low_threshold_edge_found = 1; 377 if ({ y2 < ap.YSTEP } && { ap.save_edge_detect == 1 }) 378 { 379 assert((x2+leftx) < ap.PIXX); 380 edge_image[x2+leftx][y2] = GREYSCALE_LOWEDGE; 381 } 382 else no_neighbours_left_count++; 383 } 384 else no_neighbours_left_count++; 385 } 386 if (no_neighbours_left_count == 9) 387 dir[x1][y1] = (unsigned char) </pre>		

Fringe Analysis	./fran/main.c	Page 5
<pre> 285 sobel_mode of 0 indicates a standard sobel with threshold 286 of high_threshold */ 287 minx = 1; miny = 1; 288 maxx = minx+ap.TILESIZE; 289 maxy = miny+ap.TILESIZE; 290 291 for (y = miny; y < maxy; y++) 292 for (x = minx; x < maxx; x++) 293 { 294 x1 = x - minx; y1 = y - miny; 295 dir[x1][y1] = (unsigned char) 0; 296 delta1 = 297 (tile_phase[x1][y-1] + 298 (2.0*tile_phase[x1][y]) + 299 tile_phase[x1][y+1]) - 300 (tile_phase[x-1][y-1] + 301 (2.0*tile_phase[x-1][y]) + 302 tile_phase[x-1][y+1]); 303 delta2 = 304 (tile_phase[x-1][y-1] + 305 (2.0*tile_phase[x][y-1]) + 306 tile_phase[x+1][y-1]) - 307 (tile_phase[x-1][y+1] + 308 (2.0*tile_phase[x][y+1]) + 309 tile_phase[x+1][y+1]); 310 311 above_high = 0; 312 if ((fabs((double) delta1) > high_threshold) (fabs((double) delta2) > high 313 h_threshold)) 314 { 315 dir[x1][y1] = (unsigned char) ABOVE_HIGH_THRESHOLD; 316 edges_found = 1; 317 above_high = 1; 318 if ({ y1 < ap.YSTEP } && { ap.save_edge_detect == 1 }) 319 { 320 assert((x1+leftx) < ap.PIXX); 321 edge_image[x1+leftx][y1] = GREYSCALE_BLACK; 322 } 323 } 324 } 325 if (326 { above_high == 1 } 327 { (sobel_mode == 1) && 328 { (fabs((double) delta1) > low_threshold) (fabs((double) delta2) 329 > low_threshold) } 330 } 331) 332 { 333 if (delta1 > 0.0) 334 dir[x1][y1] = 335 (unsigned char) 336 (((int) dir[x1][y1]) STEP_UP_EDGE_LEFT_TO_RIGHT); 337 if (delta1 < 0.0) 338 dir[x1][y1] = 339 (unsigned char) 340 (((int) dir[x1][y1]) STEP_DOWN_EDGE_LEFT_TO_RIGHT); 341 } </pre>		

Fringe Analysis	./fran/main.c	Page 8
the following	procedure is used.	
455	A continuous edge must touch at least two sides of the 5 by 5 box.	
456	box is chosen to be 5 by 5 because the edge detection procedure so	
457	generates an edge two pixels wide. A 3 by 3 box would not be able	
458	to distinguish a termination point as any 3 set pixels would touch tw	
459	o sides	
460	of the box whether they represented a continuous edge or not.	
461	The array box_side_touch_flags records which sides of the 5 by 5	
462	box are touched by edge pixels. A count is then made of the number	
463	of sides	
464	touched.	
465	*/	
466	box_side_touch_count = 0;	
467	for(i = 0; i < 4; i++)	
468	if (box_side_touch_flags[i] == 1) box_side_touch_count++;	
469	if (box_side_touch_count < 2)	
470	{	
471	fringe_edge_end_point_count++;	
472	if ((y1 < ap.YSTEP) && (ap.save_edge_detect == 1))	
473	{	
474	assert((xi+leftx) < ap.PIXX);	
475	edge_image[xi+leftx][y1] = GREYSCALE_EDGEEND;	
476	}	
477	}	
478	return(edges_found);	
479		
480	int Find_Good_Candidate_Scan_Horizontal(h, sy, critical)	
481	HEIGHTS h;	
482	int sy;	
483	float critical;	
484	int good_candidate = -1;	
485	int i;	
486	int ok;	
487	for(i = sy; i < (ap.TILESIZE-1); i++)	
488	{	
489	ok = 1;	
490	for(j = 0; j < ap.TILESIZE; j++)	
491	{	
492	if (fabs((double) (h[j][i] - h[j][i-1])) > critical)	
493	fabs((double) (h[j][i] - h[j][i+1])) > critical)	
494	{	
495	ok = 0; break;	
496	}	
497	if (ok == 1) { good_candidate = i; break; }	
498	}	
499	return(good_candidate);	
500		
501		
502		
503		
504		
505		
506		
507		
508		
509		
510		
511		
512		
513		
514		

Fringe Analysis	./fran/main.c	Page 7
395	((int) dir[x1][y1]) NO_NEIGHBOURS_LEFT);	
396	}	
397	}	
398	} while(low_threshold_edge_found == 1);	
399		
400	fringe_edge_set_pixel_count = 0;	
401	for(y = miny; y < maxy; y++)	
402	for(x = minx; x < maxx; x++)	
403	{	
404	fringe_edge_set_pixel_count++;	
405	if (((int) dir[x1][y1] & ABOVE_HIGH_THRESHOLD) == 0)	
406	dir[x1][y1] = (unsigned char) 0;	
407	}	
408		
409	x1 = x - minx; y1 = y - miny;	
410	if (((int) dir[x1][y1]) != 0)	
411	{	
412	fringe_edge_set_pixel_count++;	
413	if (((int) dir[x1][y1] & ABOVE_HIGH_THRESHOLD) == 0)	
414	dir[x1][y1] = (unsigned char) 0;	
415	}	
416		
417		
418		
419	fringe_edge_end_point_count = 0;	
420	for(y = miny+2; y < maxy-2; y++)	
421	for(x = minx+2; x < maxx-2; x++)	
422	{	
423	x1 = x - minx; y1 = y - miny;	
424	if (((int) dir[x1][y1]) != 0)	
425	{	
426	for(i = 0; i < 4; i++)	
427	box_side_touch_flags[i] = 0;	
428	for(y2 = y1 - 2; y2 <= y1 + 2; y2++)	
429	for(x2 = x1 - 2; x2 <= x1 + 2; x2++)	
430	{	
431	if (x2 >= 0 && (x2 < (maxx - minx)) &&	
432	((x2 < leftx) < ap.PIXX) &&	
433	y2 >= 0 && (y2 < (maxy - miny)) &&	
434	((int) dir[x2][y2]) != 0)	
435	{	
436	if (y2 == (y1 - 2)) box_side_touch_flags[TOUCHES	
437	_TOP] = 1;	
438	else if (y2 == (y1 + 2)) box_side_touch_flags[
439	TOUCHES_BOTTOM] = 1;	
440	if (x2 == (x1 - 2)) box_side_touch_flags[TOUCHES	
441	_LEFT] = 1;	
442	else if (x2 == (x1 + 2)) box_side_touch_flags[
443	TOUCHES_RIGHT] = 1;	
444	}	
445	}	
446		
447		
448		
449	/*	
450	The above code checks a 5 times 5 pixel box around each edge point	
451	fringes terminate.	
452	Such terminations suggest that the data is in some way distorted	
453	in the vicinity of that position.	
454	In order to distinguish the end of an edge from a continuous edge	

```

515 int Find_Good_Candidate_Scan_Vertical( h, sx, critical )
516 HEIGHTS h;
517 int sx;
518 float critical;
519 {
520     int good_candidate = -1;
521     int i, j;
522     int ok;
523     for( i = sx; i < (ap.TILESIZE-1); i++)
524     {
525         ok = 1;
526         for( j = 0; j < ap.TILESIZE; j++)
527             if ( (fabs( (double) ( h[i][j] - h[i-1][j] ) ) > critical) ||
528                 (fabs( (double) ( h[i][j] - h[i+1][j] ) ) > critical) )
529                 { ok = 0; break; }
530     }
531     if ( ok == 1 ) { good_candidate = i; break; }
532     return( good_candidate );
533 }
534
535 void Arrange_Vertical_Scans_Using_Good_Horizontal(hh,vh, fh, goodv)
536 HEIGHTS hh;
537 HEIGHTS vh;
538 HEIGHTS fh;
539 int goodv;
540 {
541     float diff;
542     float start_height;
543     int i, j;
544     for( i = 0; i < ap.TILESIZE; i++)
545     {
546         diff = vh[i][goodv] - hh[i][goodv];
547         for( j = 0; j < ap.TILESIZE; j++)
548             fh[i][j] = vh[i][j] - diff;
549     }
550     start_height = fh[0][0];
551     for( i = 0; i < ap.TILESIZE; i++)
552     {
553         for( j = 0; j < ap.TILESIZE; j++)
554             fh[i][j] -= start_height;
555     }
556 }
557
558 void Arrange_Horizontal_Scans_Using_Good_Vertical(hh,vh, fh, goodv)
559 HEIGHTS hh;
560 HEIGHTS vh;
561 HEIGHTS fh;
562 int goodv;
563 {
564     float diff;
565     float start_height;
566     int i, j;
567     for( i = 0; i < ap.TILESIZE; i++)
568     {
569         diff = hh[goodv][i] - vh[goodv][i];
570         for( j = 0; j < ap.TILESIZE; j++)
571             fh[j][i] = hh[j][i] - diff;
572     }
573 }
574
575 int xint = Comp_Edge(ap.xv[0], ap.yv[0], ap.xv[ap.nvertex-1], ap.yv[ap.nvertex-1], y)?

```

```

581 }
582 start_height = fh[0][0];
583 for( i = 0; i < ap.TILESIZE; i++)
584 {
585     for( j = 0; j < ap.TILESIZE; j++)
586         fh[i][j] -= start_height;
587 }
588
589 int Comp_Edge( x1,y1,x2,y2,ycrit )
590 {
591     int x1,y1,x2,y2,ycrit;
592     int tx,ty;
593     int x = -1;
594     if ( ( ycrit == y1 ) || ( ycrit == y2 ) )
595     {
596         if ( ycrit == y1 ) x = x1;
597         else x = x2;
598     }
599     else
600     {
601         if ( y1 > y2 ) { tx = x1; ty = y1; x1 = x2; y1 = y2; x2 = tx; y2 = ty; }
602         if ( ( ycrit >= y1 && ycrit <= y2 ) )
603             double m;
604             double c;
605             if ( y1 == y2 )
606             {
607                 if ( x1 > x2 ) x = x1;
608                 else x = x2;
609             }
610             else
611             {
612                 if ( x1 == x2 ) x = x1;
613                 else
614                 {
615                     m = ((double) (y2 - y1))/((double) (x2 - x1));
616                     c = ((double)y1) - (m*((double)x1));
617                     x = (int) ( ( (double) ycrit) - c ) / m + 0.5 );
618                 }
619             }
620             return( x );
621         }
622     }
623     int Point_Inside_Boundary( x,y )
624     int x,y;
625     int i;
626     int mint = 0;
627     int maxt;
628     int verdict;
629     for( i = 1; i < ap.nvertex; i++)
630     {
631         xint = Comp_Edge(ap.xv[i-1], ap.yv[i-1], ap.xv[i], ap.yv[i], y);
632         if ( ( xint != -1 ) && ( xint < x ) ) mint++;
633     }
634     xint = Comp_Edge(ap.xv[0], ap.yv[0], ap.xv[ap.nvertex-1], ap.yv[ap.nvertex-1], y);
635     if ( ( xint != -1 ) && ( xint < x ) ) mint++;
636 }

```

Fringe Analysis	./fran/main.c	Page 11
647	if (xint != -1) && (xint < x)) nint++;	
648	verdict = nint % 2;	
649	return(verdict);	
650	}	
651	int Tile_Inside_Boundary(x,y)	
652	{	
653	int verdict = 0;	
654	Point_Inside_Boundary((int) (x*(ap.XSTEP/2)), (int) (y*(ap.YSTEP/2)))	
655	{	
656	verdict =	
657	Point_Inside_Boundary((int) (x*(ap.XSTEP/2)), (int) (y*(ap.YSTEP/2)))	
658	{	
659	return(verdict);	
660	}	
661	void Fran_Error(module, error)	
662	char* module;	
663	char* error;	
664	{	
665	int cr	
666	#ifdef turboc	
667	fcloseall();	
668	#endif	
669	clrscr();	
670	printf("Error : %s\n", error);	
671		
672	if (module != NULL)	
673	printf("Module : %s\n", module);	
674		
675	printf("Do you want instructions on how to run fran (Y/N) ?\n");	
676	#ifdef unixc	
677	refresh();	
678	getch();	
679	clrscr();	
680	while ((c = toupper(getch())) != 'Y' && c != 'N');	
681	if (c == 'Y') {	
682	printf("\n");	
683	printf("How To Run The Fringe Analysis Package\n");	
684	printf("=====\n");	
685	printf("\n");	
686	printf("The package's various functions are controlled by a configuration file.\n");	
687	printf("=====\n");	
688	printf("The name of this file should be supplied as argument when the package\n");	
689	printf("is run. For example:\n");	
690	printf("./fran configuration\n");	
691	printf("The format of the configuration file is outlined below:\n\n");	
692	printf("\n");	
693	printf("Press any key to continue\n");	
694		

Fringe Analysis	./fran/main.c	Page 12
713	#ifdef unixc	
714	refresh();	
715	#endif	
716	(void) getch();	
717	clrscr();	
718		
719	printf("Configuration File Format\n");	
720	printf("=====\n");	
721	printf("\n");	
722	printf("1 : PHASE STEP [phase step] or FFT [carrier freq] or FFT_USING [raster n]	
723	or READY_WRAPPED\n");	
724	printf("\n");	
725	printf("2 : TILE_SIZE [n]\n");	
726	printf("3 : BLURR [n] or MEDIAN [n]\n");	
727	printf("4 : MOD_PERCENT [n] (where n is a floating point number)\n");	
728	printf("5 : Mode NORMALISE_ON or NORMALISE_OFF\n");	
729	printf("6 : Mode SAVE_TAN_ON or SAVE_TAN_OFF\n");	
730	printf("7 : Mode SAVE_EDGES_ON or SAVE_EDGES_OFF\n");	
731	printf("8 : Mode SAVE_LOW_MOD_ON or SAVE_LOW_MOD_OFF\n");	
732	printf("9 : Mode SAVE_GREY_ON or SAVE_GREY_OFF\n");	
733	printf("10 : Mode SAVE_TREE_ON or SAVE_TREE_OFF\n");	
734	printf("Press any key to continue\n");	
735	#ifdef unixc	
736	refresh();	
737	#endif	
738	(void) getch();	
739	clrscr();	
740	printf("11 : Mode SAVE_FFTS [start scan], [end scan] or SAVE_FFTS_OFF\n");	
741	printf("12 : Mode SINGLE_TAN or DOUBLE_TAN\n");	
742	printf("13 : Mode RAW DATA or FRINGE_COUNT or\n");	
743	printf("POLY_CORRECT n or POLY_SMOOTH n\n");	
744	printf("14 : GRID [n] (grid size of mesh in pixels)\n");	
745	printf("15 : OUT_FILES [Output file(s)* prefix]\n");	
746	printf("16 : Mode USE_RAM or USE_DISC\n");	
747	printf("17 : BOUNDS FULLFIELD or [frame] (file containing boundary definition\n");	
748	printf("of region to be processed)\n");	
749	printf("18 : [frame] (File name of image 1)\n");	
750	printf("19 : [frame] (File name of image 2 - not necessary for FFT method)\n");	
751	printf("20 : [frame] (File name of image 3 - not necessary for FFT method)\n");	
752	printf("Press any key to continue\n");	
753		

Fringe Analysis	./fran/main.c	Page 14
844	ap.nvertex = 4;	
845	{	
846	else	
847	if (fopens(frame, "r"))	
848	{	
849	if (in != NULL)	
850	do {	
851	ret = fgets(inl, 80, in);	
852	if (ret != NULL) {	
853	ap.xv[1] = atoi(inl);	
854		
855	if (ap.xv[1] < 0	
856	ap.xv[1] > (ap.PIXX + (ap.XSTEP/2)))	
857	Fran_Error(fran_module,	
858	"X coordinate in polygon boundary definition out of range");	
859		
860	ret = fgets(inl, 80, in);	
861	if (ret != NULL) {	
862	ap.yv[1] = atoi(inl);	
863		
864	if (ap.yv[1] < 0	
865	ap.yv[1] > (ap.PIXY + (ap.YSTEP/2)))	
866	Fran_Error(fran_module,	
867	"Y coordinate in polygon boundary definition out of range");	
868		
869	i++;	
870	} while(ret != NULL && i < MAXVERT);	
871		
872	if (i == MAXVERT)	
873	{	
874	printf("Too many points in boundary definition MAX = %d\n", MAXVERT);	
875		
876	ap.nvertex = i;	
877		
878	fclose(in);	
879	} else {	
880		
881	} Fran_Error(fran_module, "Couldn't open boundary definition file!");	
882		
883		
884		
885	void InitialiseAnalysis_Parameters(p)	
886	ANALYSIS_PARAMETERS* p;	
887	{	
888	strcpy(p->files_prefix, "result");	
889	strcpy(p->file_names[0], "input1.img");	
890	strcpy(p->file_names[1], "input1.img");	
891	strcpy(p->file_names[2], "input2.img");	
892	strcpy(p->config_fname, "fran.cfg");	
893	strcpy(p->bounds_fname, "FULLFIELD");	
894	p->phase_step = 0.0;	
895	p->carrier_freq = 1.0;	
896	p->ASPECT = 1.0;	
897	p->CRIT_THRES_VAL = 0.0;	
898	p->PIXX = 512;	
899	p->PIXY = 512;	
900	p->gridx = (float) (ap.PIXX / 100);	
901	p->gridy = (float) (ap.PIXY / 100);	
902	p->threshold = (float) (4.0*PIR);	
903	p->blurz = 0;	
904	p->edge_detect = 1; /* Always use adaptive thresholding. */	
905	p->median = 0;	
906	p->carrier_raster = 0;	
907	p->is_fringe_doubling = 0;	
908	p->is_phase_stepping = 0;	
909		

Fringe Analysis	./fran/main.c	Page 13
778	printf("Mesh data files will have the extension .dat. The tan fringe image\n");	
779	printf("will have the extension .tan. The modulation image will have the extension\n");	
780	};	
781	};	
782	};	
783	};	
784	};	
785	};	
786	};	
787	};	
788	};	
789	};	
790	};	
791	};	
792	};	
793	};	
794	};	
795	};	
796	};	
797	};	
798	};	
799	};	
800	};	
801	};	
802	};	
803	};	
804	};	
805	};	
806	};	
807	};	
808	};	
809	};	
810	};	
811	};	
812	};	
813	};	
814	};	
815	};	
816	};	
817	};	
818	};	
819	};	
820	};	
821	};	
822	};	
823	};	
824	};	
825	};	
826	};	
827	};	
828	};	
829	};	
830	};	
831	};	
832	};	
833	};	
834	};	
835	};	
836	};	
837	};	
838	};	
839	};	
840	};	
841	};	
842	};	
843	};	

Fringe Analysis	./fran/main.c	Page 17
1042	if (windowing == 0) {	
1043	prin("\n");	
1044	prin("Reading Configuration File: %s\n", ap.config_fname);	
1045	}	
1046	}	
1047	}	
1048	#ifdef unixc	
1049	refresh();	
1050	#endif	
1051	clear();	
1052	}	
1053	#ifdef turboc	
1054	Draw_Border();	
1055	}	
1056	}	
1057	}	
1058	#endif	
1059	printat(4, 5);	
1060	}	
1061	fgets(iline,80,config); Dcr(iline);	
1062	printf(name, "READY_WRAPPED");	
1063	}	
1064	i = 0;	
1065	do {	
1066	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1067	i++;	
1068	} while(1);	
1069	}	
1070	if (name[i] != '\0')	
1071	{	
1072	printf(name, "PHASE_STEP");	
1073	}	
1074	i = 0;	
1075	do {	
1076	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1077	i++;	
1078	} while(1);	
1079	}	
1080	if (name[i] != '\0')	
1081	{	
1082	printf(name, "FFT_USING");	
1083	}	
1084	i = 0;	
1085	do {	
1086	if ((name[i] == '\0') (name[i] != iline[i]))	
1087	{	
1088	break;	
1089	i++;	
1090	} while(1);	
1091	}	
1092	}	
1093	if (name[i] != '\0')	
1094	{	
1095	printf(name, "FFT");	
1096	}	
1097	i = 0;	
1098	do {	
1099	if ((name[i] == '\0') (name[i] != iline[i]))	
1100	{	
1101	break;	
1102	i++;	
1103	} while(1);	
1104	}	
1105	}	
1106	i++;	

Fringe Analysis	./fran/main.c	Page 18
1107	} while(1);	
1108	}	
1109	if (name[i] != '\0')	
1110	{	
1111	Frm_Error(frm_module, "PHASE_STEP (phase s	
1112	tep) or FFT (carrier freq) or FFT_USING [aster n] or READY_WRAPPED expected");	
1113	else	
1114	{	
1115	ap.carrier_freq = atof((char*)(iline + 3));	
1116	ap.phase_stepping = 0;	
1117	prin("Wrapped Phase Map Produced by : FFT Me	
1118	thod\n");	
1119	}	
1120	}	
1121	ap.carrier_raster = (unsigned) atoi((char*)(iline +	
1122	10));	
1123	ap.is_phase_stepping = 0;	
1124	prin("Mapped Phase Map Produced by : FFT Method, Ca	
1125	rrier From Raster %d\n", ap.carrier_raster);	
1126	}	
1127	else {	
1128	ap.phase_step = atof((char*)(iline + 10));	
1129	ap.is_phase_stepping = 1;	
1130	prin("Wrapped Phase Map Produced by : Phase Stepping Metho	
1131	d\n");	
1132	}	
1133	else {	
1134	ap.is_ready_wrapped = 1;	
1135	prin("Wrapped Phase Map Produced by : External Process\n");	
1136	}	
1137	printat(6, 5);	
1138	if (ap.is_phase_stepping == 1)	
1139	{	
1140	prin("Phase Step Between Input Images : %.2f degrees\n",	
1141	ap.phase_step);	
1142	}	
1143	theta = (ap.phase_step / 180.0) * 3.141592654;	
1144	/* convert phase step to radians */	
1145	}	
1146	fgets(iline,80,config); Dcr(iline);	
1147	printf(name, "TILE_SIZE");	
1148	}	
1149	i = 0;	
1150	do {	
1151	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1152	i++;	
1153	} while(1);	
1154	}	
1155	if (name[i] != '\0')	
1156	{	
1157	Frm_Error(frm_module, "TILE SIZE n expected\n");	
1158	else ap.TILESIZE = (unsigned) atoi((char*)(iline + 9));	
1159	}	
1160	if (ap.TILESIZE < 6)	
1161	{	
1162	Frm_Error(frm_module, "Tile size specified was too small.\nThe sma	
1163	llest tile allowed is 6 pixels.\nThe tile overlap is always 4 pixels.");	
1164	}	
1165	printat(7, 5);	
1166	}	

Fringe Analysis	/fran/main.c	Page 20
1230	low_mod_tol*100.0);	
1231		
1232	ap.max_dead_pixel_count = (unsigned) ((low_mod_tol *	
1233	((float) ap.TILESIZE) * _mod_tol)	
1234	((float) ap.TILESIZE)) + 0.5);	
1235	fgets(iline,80,config); Dcr(iline);	
1236		
1237	if (strcmp(iline,"NORMALISE_ON") == 0) ap.normalise = 1;	
1238	else {	
1239	if (
1240	strcmp(iline,"NORMALISE_OFF") == 0) ap.normalise = 0;	
1241	else Error(fran_module,	
1242	"Mode NORMALISE_ON or NORMALISE_OFF expected\n"	
1243);	
1244		
1245	printat(9, 5);	
1246		
1247		
1248	if (ap.normalise == 1)	
1249	prn("Input images to be normalised : Yes\n");	
1250	else prn("Input images to be normalised : No\n");	
1251		
1252	fran_module = "Memory Allocation For Tile Sized Working Buffers";	
1253		
1254	tile_phase = Create_Phase_Array(ap.TILESIZE+2, ap.TILESIZE+2);	
1255		
1256	hh = Create_Height_Array(ap.TILESIZE, ap.TILESIZE);	
1257	vh = Create_Height_Array(ap.TILESIZE, ap.TILESIZE);	
1258	rh = Create_Height_Array(ap.TILESIZE, ap.TILESIZE);	
1259	rhv = Create_Height_Array(ap.TILESIZE, ap.TILESIZE);	
1260		
1261	bl = Create_Byte_Array(ap.TILESIZE, ap.TILESIZE);	
1262	b2 = Create_Byte_Array(ap.TILESIZE, ap.TILESIZE);	
1263		
1264	fringes_passed_hor = Allocate_Integer_Array(ap.TILESIZE);	
1265	fringes_passed_ver = Allocate_Integer_Array(ap.TILESIZE);	
1266		
1267	if (fringes_passed_hor == NULL fringes_passed_ver == NULL)	
1268	{	
1269	Free(fringes_passed_hor);	
1270	Free(fringes_passed_ver);	
1271	Free(fringes_passed_hor);	
1272	Free(fringes_passed_ver);	
1273	Fran_Error(fran_module, "Memory allocation failed on fringe count arrays\n");	
1274		
1275	fran_module = "Interpreting Configuration File";	
1276		
1277	fgets(iline,80,config); Dcr(iline);	
1278		
1279	if (strcmp(iline,"SAVE_TAN_ON") == 0) ap.save_tan = 1;	
1280	else {	
1281	if (
1282	strcmp(iline,"SAVE_TAN_OFF") == 0) ap.save_tan = 0;	
1283	else	
1284	Fran_Error(fran_module,	
1285	"Mode SAVE_TAN_ON or SAVE_TAN_OFF expected\n"	
1286);	
1287		
1288	fgets(iline,80,config); Dcr(iline);	
1289		
1290	if (strcmp(iline,"SAVE_EDGES_ON") == 0)	
1291	{	
1292	ap.save_edge_detect = 1;	
1293		
1294		

Fringe Analysis	/fran/main.c	Page 19
1167	prn("Tile Size : %d pixels\n", ap.TILESIZE);	
1168		
1169	ap.XSTEP = (ap.TILESIZE-4);	
1170	ap.YSTEP = (ap.TILESIZE-4);	
1171		
1172	fgets(iline,80,config); Dcr(iline);	
1173		
1174	sprintf(name, "BLURR");	
1175		
1176	i = 0;	
1177		
1178	do {	
1179	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1180	i++;	
1181	} while(1);	
1182		
1183	if (name[i] != '\0')	
1184	{	
1185	sprintf(name, "MEDIAN");	
1186		
1187	i = 0;	
1188		
1189	do {	
1190	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1191	i++;	
1192	} while(1);	
1193		
1194	if (name[i] != '\0')	
1195	{	
1196	Fran_Error(fran_module,	
1197	"BLURR n or MEDIAN n expected\n";	
1198	else ap.median = atoi((char*)(iline + 6));	
1199		
1200	else ap.blurr = atoi((char*)(iline + 5));	
1201		
1202	printat(8, 5);	
1203		
1204	if (ap.blurr > 0)	
1205	prn("No. of Passes for 3 x 3 Averaging Filter in Preprocessing : %d\n", ap.blurr);	
1206		
1207	if (ap.median > 0)	
1208	prn("No. of Passes for 3 x 3 Median Filter in Preprocessing : %d\n", ap.median);	
1209		
1210		
1211	fgets(iline,80,config); Dcr(iline);	
1212		
1213	sprintf(name, "MOD_PERCENT");	
1214		
1215	i = 0;	
1216		
1217	do {	
1218	if ((name[i] == '\0') (name[i] != iline[i])) break;	
1219	i++;	
1220	} while(1);	
1221		
1222	if (name[i] != '\0')	
1223	{	
1224	Fran_Error(fran_module, "MOD_PERCENT n expected\n");	
1225	else low_mod_tol = (atoi((char*)(iline + 11))) / 100.0;	
1226		
1227	printat(10, 5);	
1228		
1229	prn("Percentage of Low Modulation Points after which Tile Fails : %.2f\n",	

```

1295 }
1296 else if ( strcmp(iline,"SAVE_EDGES_OFF") == 0 )
1297     ap.save.edge_detect = 0;
1298     else Fran_Error( fran_module,
1299         "Mode SAVE_EDGES_ON or SAVE_EDGES_OFF\n"
1300     );
1301
1302 fgets(iline,80,config); Dcr(iline);
1303
1304 if ( strcmp(iline,"SAVE_LOW_MOD_ON") == 0 ) ap.save_mod = 1;
1305 else {
1306     if (
1307         strcmp(iline,"SAVE_LOW_MOD_OFF") == 0 ) ap.save_mod = 0;
1308     else
1309         Fran_Error( fran_module,
1310             "Mode SAVE_LOW_MOD_ON or SAVE_LOW_MOD_OFF expected\n"
1311         );
1312     }
1313
1314 fgets(iline,80,config); Dcr(iline);
1315
1316 if ( strcmp(iline,"SAVE_GREY_ON") == 0 ) ap.save.grey = 1;
1317 else {
1318     if (
1319         strcmp(iline,"SAVE_GREY_OFF") == 0 ) ap.save.grey = 0;
1320     else
1321         Fran_Error( fran_module,
1322             "Mode SAVE_GREY_ON or SAVE_GREY_OFF expected\n"
1323         );
1324     }
1325
1326 fgets(iline,80,config); Dcr(iline);
1327
1328 if ( strcmp(iline,"SAVE_TREE_ON") == 0 ) ap.save_tree = 1;
1329 else {
1330     if (
1331         strcmp(iline,"SAVE_TREE_OFF") == 0 ) ap.save_tree = 0;
1332     else
1333         Fran_Error( fran_module,"Mode SAVE_TREE_ON or SAVE_TREE_OFF expe
1334
1335 cred\n");
1336
1337 }
1338
1339 fgets(iline,80,config); Dcr(iline);
1340
1341 sprintf( name, "SAVE_FFTS" );
1342
1343 i = 0;
1344
1345 do {
1346     if ( (name[i] == '\0') || (name[i] != iline[i]) ) break;
1347     i++;
1348 } while ( 1 );
1349
1350 if ( name[i] != '\0' )
1351     Fran_Error( fran_module,"SAVE_FFTS [n1],[n2] expected\n");
1352 else {
1353     if ( strcmp((char*)(iline+9),"_OFF") == 0 )
1354     {
1355         dump_ffts_begin_scan = -1;
1356         dump_ffts_end_scan = -1;
1357     }
1358     else {
1359         scanf( (char*)(iline+9),
1360             "%d,%d",&dump_ffts_begin_scan,
1361             &dump_ffts_end_scan );
1362     }
1363 }
1364 }
1365 }

```

```

1360 }
1361
1362 printat( 11, 5);
1363 if ( ap.is_phase_stepping == 0 ) {
1364     if ( (dump_frts_begin_scan != -1) &&
1365          (dump_frts_end_scan != -1) ) {
1366         print("Saving FFT Spectra Data : Yes From Easter : %3d To Ra
1367 ster : %3d\n",
1368             dump_frts_begin_scan, dump_frts_end_scan );
1369     }
1370     else
1371         print( "Not Saving FFT Spectra Data\n" );
1372 }
1373
1374 fgets(iline,80,config); Dcr(iline);
1375
1376 if ( strcmp(iline,"DOUBLE TAN") == 0 )
1377     ap.is_fringe_doubling = 1;
1378 else if ( strcmp(iline,"SINGLE TAN") == 0 )
1379     ap.is_fringe_doubling = 0;
1380 else
1381     ap.is_fringe_doubling = 0;
1382
1383 Fran_Error( fran_module,"Mode DOUBLE TAN or SINGLE TAN expected\n" );
1384 }
1385
1386 printat( 12, 5);
1387
1388 ap.order_of_polyfit = 0;
1389
1390 fgets(iline,80,config); Dcr(iline);
1391
1392 sprintf( name, "POLY_SMOOTH" );
1393
1394 i = 0;
1395
1396 do {
1397     if ( (name[i] == '\0') || (name[i] != iline[i]) ) break;
1398     i++;
1399 } while( 1 );
1400
1401 if ( name[i] != '\0' )
1402 {
1403     sprintf( name, "POLY_CORRECT" );
1404     i = 0;
1405
1406     do {
1407         if ( (name[i] == '\0') || (name[i] != iline[i]) ) break;
1408         i++;
1409     } while( 1 );
1410
1411     if ( name[i] != '\0' )
1412     {
1413         if ( strcmp(iline,"RAW_DATA") == 0 )
1414             print("Mode of Tile Unwrapping : From Raw Data via MST Pixel Method\n");
1415         ap.unwrap_tile_by_mst = 1;
1416     }
1417     else
1418     {
1419         if ( strcmp(iline,"FRINGE_COUNT") == 0 )
1420             print("Mode of Tile Unwrapping : From Raw Data via Fringe Counting Method\n");
1421     }
1422 }
1423
1424 ap.unwrap_tile_by_mst = 0;

```

Fringe Analysis	./fran/main.c	Page 24
1485	prn("Edge Detection of Fringes File : %s.edg\n", ap.files_prefix);	
1486	else	
1487	prn("Edge Detection of Fringes File : NOT SAVED\n");	
1488	printat(17, 5);	
1489	if (ap.save_mod == 1)	
1490	prn("Points of Low Modulation File : %s.mod\n", ap.files_prefix);	
1491	else	
1492	prn("Points of Low Modulation File : NOT SAVED\n");	
1493	printat(18, 5);	
1494	if (ap.save_gray == 1)	
1495	prn("Gray Scale Height Map File : %s.gray\n", ap.files_prefix);	
1496	else	
1497	prn("Gray Scale Height Map File : NOT SAVED\n");	
1498	printat(19, 5);	
1499	if (ap.save_tree == 1)	
1500	prn("File Assembly Tree File : %s.tree\n", ap.files_prefix);	
1501	else	
1502	prn("File Assembly Tree File : NOT SAVED\n");	
1503	fgets(iline, 80, config); Dcr(iline);	
1504	if (strcmp(iline, "USE_RAM") == 0) ap.use_ram = 1;	
1505	else if (strcmp(iline, "USE_DISC") == 0)	
1506	{	
1507	ap.use_ram = 0;	
1508	}	
1509	else Fran_Error(fran_module,	
1510	"Mode USE_RAM or USE_DISC expected\n"	
1511);	
1512	/* Force use of disc */	
1513	ap.use_ram = 0;	
1514	/* Force use of disc */	
1515	ap.use_ram = 0;	
1516	/* Force use of disc */	
1517	ap.use_ram = 0;	
1518	/* Force use of disc */	
1519	ap.use_ram = 0;	
1520	/* Force use of disc */	
1521	ap.use_ram = 0;	
1522	/* Force use of disc */	
1523	ap.use_ram = 0;	
1524	/* Force use of disc */	
1525	ap.use_ram = 0;	
1526	/* Force use of disc */	
1527	ap.use_ram = 0;	
1528	/* Force use of disc */	
1529	ap.use_ram = 0;	
1530	/* Force use of disc */	
1531	ap.use_ram = 0;	
1532	/* Force use of disc */	
1533	ap.use_ram = 0;	
1534	/* Force use of disc */	
1535	ap.use_ram = 0;	
1536	/* Force use of disc */	
1537	ap.use_ram = 0;	
1538	/* Force use of disc */	
1539	ap.use_ram = 0;	
1540	/* Force use of disc */	
1541	ap.use_ram = 0;	
1542	/* Force use of disc */	
1543	ap.use_ram = 0;	
1544	/* Force use of disc */	
1545	ap.use_ram = 0;	
1546	/* Force use of disc */	
1547	ap.use_ram = 0;	
1548	/* Force use of disc */	
1549	ap.use_ram = 0;	
1550	/* Force use of disc */	

Fringe Analysis	./fran/main.c	Page 23
1423	else Fran_Error(fran_module,	
1424	"Mode RAW_DATA or FRINGE_COUNT or POLY_SMOOTH n or POLY_CORRECT n expected\n"	
1425);	
1426	else {	
1427	ap.smooth = 0;	
1428	ap.order_of_polyfit = (unsigned) atoi((char*)(iline + 12));	
1429	ap.order_of_polyfit = (unsigned) atoi((char*)(iline + 12));	
1430	ap.order_of_polyfit = (unsigned) atoi((char*)(iline + 12));	
1431	ap.order_of_polyfit = (unsigned) atoi((char*)(iline + 12));	
1432	prn("Mode of Tile Unwrapping : Fringe Counting with %d Degree Poly Correct	
1433	ion Fit\n",	
1434	ap.order_of_polyfit);	
1435	ap.unwrap_tile_by_mat = 0;	
1436	ap.unwrap_tile_by_mat = 0;	
1437	ap.unwrap_tile_by_mat = 0;	
1438	ap.unwrap_tile_by_mat = 0;	
1439	ap.unwrap_tile_by_mat = 0;	
1440	ap.unwrap_tile_by_mat = 0;	
1441	ap.unwrap_tile_by_mat = 0;	
1442	ap.unwrap_tile_by_mat = 0;	
1443	ap.unwrap_tile_by_mat = 0;	
1444	ap.unwrap_tile_by_mat = 0;	
1445	ap.unwrap_tile_by_mat = 0;	
1446	ap.unwrap_tile_by_mat = 0;	
1447	ap.unwrap_tile_by_mat = 0;	
1448	ap.unwrap_tile_by_mat = 0;	
1449	ap.unwrap_tile_by_mat = 0;	
1450	ap.unwrap_tile_by_mat = 0;	
1451	ap.unwrap_tile_by_mat = 0;	
1452	ap.unwrap_tile_by_mat = 0;	
1453	ap.unwrap_tile_by_mat = 0;	
1454	ap.unwrap_tile_by_mat = 0;	
1455	ap.unwrap_tile_by_mat = 0;	
1456	ap.unwrap_tile_by_mat = 0;	
1457	ap.unwrap_tile_by_mat = 0;	
1458	ap.unwrap_tile_by_mat = 0;	
1459	ap.unwrap_tile_by_mat = 0;	
1460	ap.unwrap_tile_by_mat = 0;	
1461	ap.unwrap_tile_by_mat = 0;	
1462	ap.unwrap_tile_by_mat = 0;	
1463	ap.unwrap_tile_by_mat = 0;	
1464	ap.unwrap_tile_by_mat = 0;	
1465	ap.unwrap_tile_by_mat = 0;	
1466	ap.unwrap_tile_by_mat = 0;	
1467	ap.unwrap_tile_by_mat = 0;	
1468	ap.unwrap_tile_by_mat = 0;	
1469	ap.unwrap_tile_by_mat = 0;	
1470	ap.unwrap_tile_by_mat = 0;	
1471	ap.unwrap_tile_by_mat = 0;	
1472	ap.unwrap_tile_by_mat = 0;	
1473	ap.unwrap_tile_by_mat = 0;	
1474	ap.unwrap_tile_by_mat = 0;	
1475	ap.unwrap_tile_by_mat = 0;	
1476	ap.unwrap_tile_by_mat = 0;	
1477	ap.unwrap_tile_by_mat = 0;	
1478	ap.unwrap_tile_by_mat = 0;	
1479	ap.unwrap_tile_by_mat = 0;	
1480	ap.unwrap_tile_by_mat = 0;	
1481	ap.unwrap_tile_by_mat = 0;	
1482	ap.unwrap_tile_by_mat = 0;	
1483	ap.unwrap_tile_by_mat = 0;	
1484	ap.unwrap_tile_by_mat = 0;	

```

1551 if ( ap.is_phase_stepping == 1 ) {
1552     for( i = 0; i < 3; i++ )
1553     {
1554         fgets(ap.file_names[i],80,config); Dcr(ap.file_names[i]);
1555     }
1556     else {
1557         fgets(ap.file_names[0],80,config); Dcr(ap.file_names[0]);
1558     }
1559     }
1560     }
1561     }
1562     }
1563     }
1564     }
1565     }
1566     }
1567     }
1568     }
1569     }
1570     }
1571     }
1572     }
1573     }
1574     }
1575     }
1576     }
1577     }
1578     }
1579     }
1580     }
1581     }
1582     }
1583     }
1584     }
1585     }
1586     }
1587     }
1588     }
1589     }
1590     }
1591     }
1592     }
1593     }
1594     }
1595     }
1596     }
1597     }
1598     }
1599     }
1600     }
1601     }
1602     }
1603     }
1604     }
1605     }
1606     }
1607     }
1608     }
1609     }
1610     }
1611     }
1612     }

```

Fringe Analysis

./fran/main.c

Page 25

Fringe Analysis

./fran/main.c

Page 26

```

1613 }
1614 }
1615 }
1616 }
1617 }
1618 }
1619 }
1620 }
1621 }
1622 }
1623 }
1624 }
1625 }
1626 }
1627 }
1628 }
1629 }
1630 }
1631 }
1632 }
1633 }
1634 }
1635 }
1636 }
1637 }
1638 }
1639 }
1640 }
1641 }
1642 }
1643 }
1644 }
1645 }
1646 }
1647 }
1648 }
1649 }
1650 }
1651 }
1652 }
1653 }
1654 }
1655 }
1656 }
1657 }
1658 }
1659 }
1660 }
1661 }
1662 }
1663 }
1664 }
1665 }
1666 }
1667 }
1668 }
1669 }
1670 }
1671 }
1672 }
1673 }

```


Fringe Analysis	./fran/main.c	Page 30
1869	else dotile = 1;	
1870		
1871	if (
1872	{ dotile = 1 } &&	
1873	{ Tile_Inside_Boundary(leftx, topy) == 1 }	
1874	})	
1875		
1876	{	
1877	Generate_Tile_Phase_Array(tile_phase);	
1878	edge_data = bl;	
1879		
1880	any_edges = Sobel_Edge_Detect_With_Hysteresis_Thresholding	
1881	(tile_phase, ap.edge_detect,	
1882	{ 2.0 * ap.threshold / 3.0 }, ap.threshold, edge_data);	
1883		
1884	if (Mark_Dead_Pixels(tile_phase, ap.max_dead_pixel_count) == 1)	
1885	{	
1886		
1887	fran_module = "Unwrapping a Tile";	
1888		
1889	if (any_edges == 0)	
1890	{	
1891	Create_Tile_Array_Element_Directly_From_Phase	
1892	(ntx, nty, tile_phase, dead_pixel_count, ap.use_ram);	
1893		
1894	else {	
1895		
1896	if (ap.unwrap_tile_by_mst == 1)	
1897	{	
1898	Unwrap_Tile_Using_Mst(edge_data, hh, tile_phase);	
1899		
1900	Create_Tile_Array_Element	
1901	(ntx, nty, hh, "m", 0, 0, fringe_edge_set_pixel_count, fringe_edge_end_point_count,	
1902	t_dead_pixel_count, ap.use_ram);	
1903		
1904		
1905	confidence_hor =	
1906	Unwrap_Horizontal_Scans(edge_data, hh, fringes_passed_hor, tile_phase);	
1907		
1908	if (confidence_hor != 0.0)	
1909	confidence_ver = Unwrap_Vertical_Scans(edge_data,	
1910	confidence_ver != 0.0);	
1911	else confidence_ver = 0.0;	
1912		
1913	if ((confidence_hor != 0.0) (confidence_ver != 0.0)) {	
1914		
1915	int dont_use_hor = 0;	
1916	int dont_use_ver = 0;	
1917		
1918	if (confidence_hor != 0.0)	
1919	goodh = Find_Good_Candidate_Scan_Horizontal(hh, 1, DELTA_FOR_EDGE);	
1920	else goodh = -1;	
1921	if (confidence_ver != 0.0)	
1922	goodv = Find_Good_Candidate_Scan_Vertical(vh, 1, DELTA_FOR_EDGE);	
1923	else goodv = -1;	
1924		
1925	if ((confidence_hor == 0.0) (goodv == -1))	
1926	int dont_use_hor = 1;	
1927		
1928	if ((confidence_ver == 0.0) (goodh == -1))	
1929	int dont_use_ver = 1;	
1930		
1931	if ((confidence_ver == 0.0) (goodh == -1))	
1932	int dont_use_ver = 1;	
1933		

Fringe Analysis	./fran/main.c	Page 29
1803	fran_module = "Clearing Row Memory Buffer for Fringe Edge Detection";	
1804		
1805	for(ye = 0; ye < ap.YSTEP; ye++)	
1806	for(xe = 0; xe < ap.PIXX; xe++) edge_image[xe][ye] = GREYSCALE_WHITE;	
1807		
1808	for(xe = 0; xe < ap.PIXX; xe++) edge_image[xe][0] = GREYSCALE_GREY;	
1809		
1810	for(xe = 0; xe < ap.PIXX; xe += ap.XSTEP)	
1811	for(ye = 0; ye < ap.YSTEP; ye++) edge_image[xe][ye] = GREYSCALE_GREY;	
1812		
1813	for(ye = 0; ye < ap.YSTEP; ye++)	
1814	{	
1815	edge_image[0][ye] = GREYSCALE_GREY;	
1816	edge_image[ap.PIXX-1][ye] = GREYSCALE_GREY;	
1817		
1818		
1819		
1820		
1821	fran_module = "Computing Tile Sized Row of Wrapped Map";	
1822		
1823	if (ap.is_ready_wrapped == 1)	
1824	{	
1825	Compute_Wrapped_Row_Of_Tan_Fringes	
1826	(ap.save_tan);	
1827		
1828	else {	
1829		
1830	if (ap.is_phase_stepping == 1)	
1831	{	
1832	Compute_Phase_Step_Row_Of_Tan_Fringes	
1833	(ap.save_tan, ap.save_mod, ap.is_fringe_doubling);	
1834		
1835	else {	
1836		
1837	cutoff_print_count = 1;	
1838	Compute_FFT_Row_Of_Tan_Fringes	
1839	(ap.save_tan, ap.save_mod, ap.is_fringe_doubling);	
1840		
1841	topy = y;	
1842	for(x = 0; x < ap.PIXX; x += ap.XSTEP)	
1843	{	
1844	int dotile;	
1845		
1846	printat(1, 25);	
1847		
1848	if (windowing == 0) {	
1849	prn("Processing Tile %4d,%4d", ntx, nty);	
1850		
1851	#ifdef unix	
1852	refresh();	
1853	#endif	
1854		
1855	leftx = x;	
1856		
1857	if ((ap.is_ready_wrapped == 0) && (ap.is_phase_stepping == 0))	
1858	{	
1859	if (threshold_results[ntx] < ((int) ap.CRIT_THRES_VAL))	
1860	dotile = 0;	
1861	else	
1862	dotile = 1;	
1863		
1864		
1865		
1866		
1867		
1868		

Fringe Analysis	./fran/main.c	Page 31
1934	if ((dont_use_hor == 0) (dont_use_ver == 0))	
1935	{	
1936	int use_hor = 0;	
1937	int use_ver = 0;	
1938	if (((dont_use_hor == 0) && (dont_use_ver == 1)))	
1939	{	
1940	use_hor = 1;	
1941	}	
1942	else {	
1943	{	
1944	if (((dont_use_ver == 0) && (dont_use_hor == 1)))	
1945	{	
1946	use_ver = 1;	
1947	}	
1948	else {	
1949	{	
1950	if (confidence_hor > confidence_ver)	
1951	{	
1952	use_hor = 1;	
1953	else use_ver = 1;	
1954	}	
1955	}	
1956	}	
1957	}	
1958	if (use_hor == 1) {	
1959	{	
1960	Arrange Horizontal Scans Using Good_Verical(hh,vh,fhh,goodv);	
1961	Create_Tile_Array_Element	
1962	(ntx,nty,fhv,h',ap.order_of_polyfit,ap.smooth,	
1963	fringe_edge_set_pixel_count,fringe_edge_end_point_count,dead_pixe	
1964	l_count,	
1965	ap.use_ram);	
1966	}	
1967	}	
1968	if (use_ver == 1) {	
1969	{	
1970	Arrange Vertical Scans Using Good_Horizontal(th,vh,fhv,goodh);	
1971	Create_Tile_Array_Element	
1972	(ntx,nty,fhv,v',ap.order_of_polyfit,ap.smooth,	
1973	fringe_edge_set_pixel_count,fringe_edge_end_point_count,dead_pixe	
1974	l_count,	
1975	ap.use_ram);	
1976	}	
1977	}	
1978	}	
1979	}	
1980	}	
1981	}	
1982	/* end of else for fringe counting option */	
1983	/* end of else for any edges */	
1984	/* end of ap_max_dead_pixel_count if */	
1985	}	
1986	}	
1987	}	
1988	ntx++;	
1989	/* end of x for loop */	
1990	if (ap.save_edge_detect == 1) {	
1991	{	
1992	for (ye = 0; ye < ap.YSTEP; ye++)	
1993	{	
1994	for (xe = 0; xe < ap.PIXX; xe++)	
1995	{	
1996	edge_raster[xe] = edge_image[xe][ye];	
1997	}	

Fringe Analysis	./fran/main.c	Page 32
1998	if (((copy + ye) < ap.PIXY)	
1999	TIFFWriteScanline(edge_out, (u_char *) edge_raster,	
2000	(u_int) (copy+ye), (u_int) 0);	
2001	}	
2002	}	
2003	}	
2004	nty++;	
2005	/* end of y for loop */	
2006	if (ap.save_edge_detect == 1) {	
2007	{	
2008	copy += ap.YSTEP;	
2009	}	
2010	if ((copy < ap.PIXY) {	
2011	{	
2012	for (xe = 0; xe < ap.PIXX; xe++) edge_raster[xe] = GREYSCALE_GREY;	
2013	/*	
2014	for (xe = 0; xe < ap.PIXX; xe++) edge_raster[xe] = GREYSCALE_GREY;	
2015	*/	
2016	TIFFWriteScanline(edge_out, (u_char *) edge_raster, (u_int) (copy), (u_int) 0	
2017);	
2018	for (xe = 0; xe < ap.PIXX; xe++) edge_raster[xe] = GREYSCALE_WHITE;	
2019	for (ye = (copy+1); ye < ap.PIXY; ye++)	
2020	TIFFWriteScanline(edge_out, (u_char *) edge_raster, (u_int) (ye), (u_int) 0);	
2021	}	
2022	}	
2023	}	
2024	}	
2025	Free_Array(edge_image, ap.PIXX);	
2026	TIFFFlushData(edge_out);	
2027	TIFFClose(edge_out);	
2028	Free(edge_raster);	
2029	}	
2030	if ((ap.use_ram == 0) && (use_saved_solutions == 0))	
2031	{	
2032	{	
2033	frank_module = "Saving Tile Array Data other than the Unwrapped Solutions";	
2034	for (nty = 0; nty < ap.NTILES; nty++)	
2035	{	
2036	for (ntx = 0; ntx < ap.NTILES; ntx++)	
2037	{	
2038	if ((ta[ntx][nty] != NULL)	
2039	Write_Tile_Array_Element(ta[ntx][nty], ap.tile_array_dat);	
2040	}	
2041	}	
2042	}	
2043	}	
2044	}	
2045	}	
2046	clrscr();	
2047	}	
2048	else Read_Saved_Tile_Array_Data();	
2049	clrscr();	
2050	}	
2051	frank_module = "Freeing Buffers Used";	
2052	Free_Phase_Array(tile_phase, ap.TILESIZE*2);	
2053	Deallocate_Tile_Overlap_Arrays();	
2054	Free_Phase_Array(b1, ap.TILESIZE);	
2055	Free_Phase_Array(b2, ap.TILESIZE);	
2056	Free_Phase_Array(b3, ap.TILESIZE);	
2057	Free_Phase_Array(b4, ap.TILESIZE);	
2058	Free_Phase_Array(b5, ap.TILESIZE);	
2059	Free_Phase_Array(b6, ap.TILESIZE);	
2060	Free_Phase_Array(b7, ap.TILESIZE);	
2061	Free_Phase_Array(b8, ap.TILESIZE);	
2062	Free_Phase_Array(b9, ap.TILESIZE);	

Fringe Analysis	./fran/main.c	Page 33
<pre> 2063 prn("\n"); 2064 prn("Constructing Tile Arrangement Tree\n"); 2065 #ifdef unix 2066 refresh(); 2067 #endif 2068 fran_module = "Unwrapping Tile to Tile by Constructing MST"; 2069 if (ap.save_tree == 1) 2070 { 2071 printf(name, "%s.tree", ap.files_prefix); 2072 Explore(name); 2073 else Explore((char*) NULL); 2074 Free Memory Used In Mst Tile Unwrap(); 2075 fran_module = "Writing Output Tif Image Files and Numeric Data Files"; 2076 Find_Min_And_Max_Heights(minimum_height, maximum_height); 2077 if (ap.save_grey == 1) { 2078 prn("Saving grey scale height map image and mesh array...\n"); 2079 #ifdef unix 2080 refresh(); 2081 #endif 2082 printf(name, "%s.grey", ap.files_prefix); 2083 printf(name, "%s.dat", ap.files_prefix); 2084 Write_Mesh_And_Grey_Scale_Map(name, name, 2085 ap.gridx, ap.gridy, minimum_height, maximum_height, ap.use_ram); 2086 } 2087 } 2088 } 2089 } 2090 } 2091 } 2092 } 2093 } 2094 } 2095 } 2096 } 2097 } 2098 } 2099 } 2100 } 2101 } 2102 } 2103 } 2104 } 2105 } 2106 } 2107 } 2108 } 2109 } 2110 } 2111 } 2112 } 2113 } 2114 } 2115 } 2116 } 2117 } 2118 } 2119 } 2120 } 2121 } 2122 } 2123 } 2124 } 2125 } 2126 } 2127 } 2128 } </pre>	<pre> 2129 { 2130 fran_module = "Closing Files and Freeing Buffers Used"; 2131 if (ap.is_ready_wrapped == 1) 2132 { 2133 Close_Wrapped_Computation(ap.save_tan); 2134 } 2135 else { 2136 if (ap.is_phase_stepping == 1) 2137 { 2138 Close_Phase_Step_Tan_Fringe_Computation 2139 (ap.save_tan, ap.save_mod, ap.is_fringe_doubling); 2140 } 2141 else { 2142 Close_FFT_Tan_Fringe_Computation 2143 (ap.save_tan, ap.save_mod, ap.is_fringe_doubling); 2144 } 2145 } 2146 elapsed_time = (double) clock(); 2147 minutes = (int) (elapsed_time / (60.0*TICKS) +0.5); 2148 seconds = ((int) (elapsed_time / TICKS)) % 60 ; 2149 prn("\nImage processing took %d minutes %d seconds\n", minutes, seconds); 2150 #ifdef unix 2151 refresh(); 2152 #endif 2153 } else 2154 { 2155 Fran_Error(fran_module, "Configuration file not specified\n"); 2156 } 2157 } 2158 } 2159 } 2160 } 2161 } 2162 } 2163 } 2164 } 2165 } 2166 } </pre>	<pre> 2167 } 2168 } 2169 } 2170 } 2171 } 2172 } 2173 } 2174 } 2175 } 2176 } 2177 } 2178 } 2179 } 2180 } 2181 } 2182 } 2183 } 2184 } 2185 } 2186 } 2187 } 2188 } 2189 } 2190 } 2191 } 2192 } 2193 } 2194 } 2195 } 2196 } 2197 } 2198 } 2199 } 2200 } 2201 } 2202 } 2203 } 2204 } 2205 } 2206 } 2207 } 2208 } 2209 } 2210 } 2211 } 2212 } 2213 } 2214 } 2215 } 2216 } 2217 } 2218 } 2219 } 2220 } 2221 } 2222 } 2223 } 2224 } 2225 } 2226 } 2227 } 2228 } 2229 } 2230 } 2231 } 2232 } 2233 } 2234 } 2235 } 2236 } 2237 } 2238 } 2239 } 2240 } 2241 } 2242 } 2243 } 2244 } 2245 } 2246 } 2247 } 2248 } 2249 } 2250 } 2251 } 2252 } 2253 } 2254 } 2255 } 2256 } 2257 } 2258 } 2259 } 2260 } 2261 } 2262 } 2263 } 2264 } 2265 } 2266 } 2267 } 2268 } 2269 } 2270 } 2271 } 2272 } 2273 } 2274 } 2275 } 2276 } 2277 } 2278 } 2279 } 2280 } 2281 } 2282 } 2283 } 2284 } 2285 } 2286 } 2287 } 2288 } 2289 } 2290 } 2291 } 2292 } 2293 } 2294 } 2295 } 2296 } 2297 } 2298 } 2299 } 2300 } 2301 } 2302 } 2303 } 2304 } 2305 } 2306 } 2307 } 2308 } 2309 } 2310 } 2311 } 2312 } 2313 } 2314 } 2315 } 2316 } 2317 } 2318 } 2319 } 2320 } 2321 } 2322 } 2323 } 2324 } 2325 } 2326 } 2327 } 2328 } 2329 } 2330 } 2331 } 2332 } 2333 } 2334 } 2335 } 2336 } 2337 } 2338 } 2339 } 2340 } 2341 } 2342 } 2343 } 2344 } 2345 } 2346 } 2347 } 2348 } 2349 } 2350 } 2351 } 2352 } 2353 } 2354 } 2355 } 2356 } 2357 } 2358 } 2359 } 2360 } 2361 } 2362 } 2363 } 2364 } 2365 } 2366 } 2367 } 2368 } 2369 } 2370 } 2371 } 2372 } 2373 } 2374 } 2375 } 2376 } 2377 } 2378 } 2379 } 2380 } 2381 } 2382 } 2383 } 2384 } 2385 } 2386 } 2387 } 2388 } 2389 } 2390 } 2391 } 2392 } 2393 } 2394 } 2395 } 2396 } 2397 } 2398 } 2399 } 2400 } 2401 } 2402 } 2403 } 2404 } 2405 } 2406 } 2407 } 2408 } 2409 } 2410 } 2411 } 2412 } 2413 } 2414 } 2415 } 2416 } 2417 } 2418 } 2419 } 2420 } 2421 } 2422 } 2423 } 2424 } 2425 } 2426 } 2427 } 2428 } 2429 } 2430 } 2431 } 2432 } 2433 } 2434 } 2435 } 2436 } 2437 } 2438 } 2439 } 2440 } 2441 } 2442 } 2443 } 2444 } 2445 } 2446 } 2447 } 2448 } 2449 } 2450 } 2451 } 2452 } 2453 } 2454 } 2455 } 2456 } 2457 } 2458 } 2459 } 2460 } 2461 } 2462 } 2463 } 2464 } 2465 } 2466 } 2467 } 2468 } 2469 } 2470 } 2471 } 2472 } 2473 } 2474 } 2475 } 2476 } 2477 } 2478 } 2479 } 2480 } 2481 } 2482 } 2483 } 2484 } 2485 } 2486 } 2487 } 2488 } 2489 } 2490 } 2491 } 2492 } 2493 } 2494 } 2495 } 2496 } 2497 } 2498 } 2499 } 2500 } 2501 } 2502 } 2503 } 2504 } 2505 } 2506 } 2507 } 2508 } 2509 } 2510 } 2511 } 2512 } 2513 } 2514 } 2515 } 2516 } 2517 } 2518 } 2519 } 2520 } 2521 } 2522 } 2523 } 2524 } 2525 } 2526 } 2527 } 2528 } 2529 } 2530 } 2531 } 2532 } 2533 } 2534 } 2535 } 2536 } 2537 } 2538 } 2539 } 2540 } 2541 } 2542 } 2543 } 2544 } 2545 } 2546 } 2547 } 2548 } 2549 } 2550 } 2551 } 2552 } 2553 } 2554 } 2555 } 2556 } 2557 } 2558 } 2559 } 2560 } 2561 } 2562 } 2563 } 2564 } 2565 } 2566 } 2567 } 2568 } 2569 } 2570 } 2571 } 2572 } 2573 } 2574 } 2575 } 2576 } 2577 } 2578 } 2579 } 2580 } 2581 } 2582 } 2583 } 2584 } 2585 } 2586 } 2587 } 2588 } 2589 } 2590 } 2591 } 2592 } 2593 } 2594 } 2595 } 2596 } 2597 } 2598 } 2599 } 2600 } 2601 } 2602 } 2603 } 2604 } 2605 } 2606 } 2607 } 2608 } 2609 } 2610 } 2611 } 2612 } 2613 } 2614 } 2615 } 2616 } 2617 } 2618 } 2619 } 2620 } 2621 } 2622 } 2623 } 2624 } 2625 } 2626 } 2627 } 2628 } 2629 } 2630 } 2631 } 2632 } 2633 } 2634 } 2635 } 2636 } 2637 } 2638 } 2639 } 2640 } 2641 } 2642 } 2643 } 2644 } 2645 } 2646 } 2647 } 2648 } 2649 } 2650 } 2651 } 2652 } 2653 } 2654 } 2655 } 2656 } 2657 } 2658 } 2659 } 2660 } 2661 } 2662 } 2663 } 2664 } 2665 } 2666 } 2667 } 2668 } 2669 } 2670 } 2671 } 2672 } 2673 } 2674 } 2675 } 2676 } 2677 } 2678 } 2679 } 2680 } 2681 } 2682 } 2683 } 2684 } 2685 } 2686 } 2687 } 2688 } 2689 } 2690 } 2691 } 2692 } 2693 } 2694 } 2695 } 2696 } 2697 } 2698 } 2699 } 2700 } 2701 } 2702 } 2703 } 2704 } 2705 } 2706 } 2707 } 2708 } 2709 } 2710 } 2711 } 2712 } 2713 } 2714 } 2715 } 2716 } 2717 } 2718 } 2719 } 2720 } 2721 } 2722 } 2723 } 2724 } 2725 } 2726 } 2727 } 2728 } 2729 } 2730 } 2731 } 2732 } 2733 } 2734 } 2735 } 2736 } 2737 } 2738 } 2739 } 2740 } 2741 } 2742 } 2743 } 2744 } 2745 } 2746 } 2747 } 2748 } 2749 } 2750 } 2751 } 2752 } 2753 } 2754 } 2755 } 2756 } 2757 } 2758 } 2759 } 2760 } 2761 } 2762 } 2763 } 2764 } 2765 } 2766 } 2767 } 2768 } 2769 } 2770 } 2771 } 2772 } 2773 } 2774 } 2775 } 2776 } 2777 } 2778 } 2779 } 2780 } 2781 } 2782 } 2783 } 2784 } 2785 } 2786 } 2787 } 2788 } 2789 } 2790 } 2791 } 2792 } 2793 } 2794 } 2795 } 2796 } 2797 } 2798 } 2799 } 2800 } 2801 } 2802 } 2803 } 2804 } 2805 } 2806 } 2807 } 2808 } 2809 } 2810 } 2811 } 2812 } 2813 } 2814 } 2815 } 2816 } 2817 } 2818 } 2819 } 2820 } 2821 } 2822 } 2823 } 2824 } 2825 } 2826 } 2827 } 2828 } 2829 } 2830 } 2831 } 2832 } 2833 } 2834 } 2835 } 2836 } 2837 } 2838 } 2839 } 2840 } 2841 } 2842 } 2843 } 2844 } 2845 } 2846 } 2847 } 2848 } 2849 } 2850 } 2851 } 2852 } 2853 } 2854 } 2855 } 2856 } 2857 } 2858 } 2859 } 2860 } 2861 } 2862 } 2863 } 2864 } 2865 } 2866 } 2867 } 2868 } 2869 } 2870 } 2871 } 2872 } 2873 } 2874 } 2875 } 2876 } 2877 } 2878 } 2879 } 2880 } 2881 } 2882 } 2883 } 2884 } 2885 } 2886 } 2887 } 2888 } 2889 } 2890 } 2891 } 2892 } 2893 } 2894 } 2895 } 2896 } 2897 } 2898 } 2899 } 2900 } 2901 } 2902 } 2903 } 2904 } 2905 } 2906 } 2907 } 2908 } 2909 } 2910 } 2911 } 2912 } 2913 } 2914 } 2915 } 2916 } 2917 } 2918 } 2919 } 2920 } 2921 } 2922 } 2923 } 2924 } 2925 } 2926 } 2927 } 2928 } 2929 } 2930 } 2931 } 2932 } 2933 } 2934 } 2935 } 2936 } 2937 } 2938 } 2939 } 2940 } 2941 } 2942 } 2943 } 2944 } 2945 } 2946 } 2947 } 2948 } 2949 } 2950 } 2951 } 2952 } 2953 } 2954 } 2955 } 2956 } 2957 } 2958 } 2959 } 2960 } 2961 } 2962 } 2963 } 2964 } 2965 } 2966 } 2967 } 2968 } 2969 } 2970 } 2971 } 2972 } 2973 } 2974 } 2975 } 2976 } 2977 } 2978 } 2979 } 2980 } 2981 } 2982 } 2983 } 2984 } 2985 } 2986 } 2987 } 2988 } 2989 } 2990 } 2991 } 2992 } 2993 } 2994 } 2995 } 2996 } 2997 } 2998 } 2999 } 3000 } </pre>
Fringe Analysis	./fran/main.c	Page 34

Fringe Analysis	./fran/polysmooth.h	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	extern void Poly_Smooth_Verically();	
6	extern void Poly_Smooth_Horizontally();	
7	extern void Poly_Correct_Verically();	
8	extern void Poly_Correct_Horizontally();	
9	extern int Locate_Side_Lobe_Cut_Off_Freq();	
10		

Fringe Analysis	./fran/polysmooth.c	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	#include <stdio.h>	
6	#include <ctype.h>	
7	#include <math.h>	
8	#include <string.h>	
9	#include <time.h>	
10	#include "franhdr.h"	
11	#define turboc	
12	#define malloc	
13	#endif	
14	#define unxc	
15	#include <malloc.h>	
16	#include <urses.h>	
17	#endif	
18	#include "tiffio.h"	
19	#include "correct.h"	
20	#include "correctoffsets.h"	
21	#include "fringecount.h"	
22	#include "dynamicarray.h"	
23	#include "imageproc.h"	
24	#include "compuwrap.h"	
25	#include "tileelements.h"	
26	#include "main.h"	
27	#include "polysmooth.h"	
28		
29	void nerror(error_text)	
30	char error_text[];	
31	{	
32	Fran_Error(fran_module, error_text);	
33	}	
34		
35	float* vector(nlo,nh)	
36	{	
37	int nlo,nh;	
38	float* v;	
39		
40	v = (float*) Malloc((unsigned) (nh*nlo+1)*sizeof(float));	
41	if (v == NULL) nerror("allocation failure in vector()");	
42	return v-nlo;	
43	}	
44		
45	void free_vector(v,nlo,nh)	
46	float* v;	
47	int nlo,nh;	
48	{	
49	Free((char*) (v-nlo));	
50	}	
51		
52	void fpoly(v,p,np)	
53	float x,p[];	
54	int np;	
55	{	
56	int j;	
57		
58	p[1]=1.0;	
59	for (j=2;j<np;j++) p[j]=p[j-1]*x;	
60	}	
61		
62	void subkab(u,w,v,m,n,b,x)	
63	float **u,w[],**v,b[];x[];	
64	int m,n;	
65	{	
66		

Fringe Analysis	./fran/polysmooth.c	Page 2
67	int j,j,i;	
68	float s,tmp,*vector();	
69	void free_vector();	
70		
71	tmp=vector(l,n);	
72	for (j=1;j<n;j++) {	
73	if (w[j]) {	
74	for (i=1;i<m;i++) s += u[i][j]*b[i];	
75	s /= w[j];	
76	tmp[j]=s;	
77	}	
78	for (j=1;j<n;j++) {	
79	s=0.0;	
80	for (j=1;j<n;j++) s += v[j][j]*tmp[j];	
81	x[j]=s;	
82	free_vector(tmp,l,n);	
83	}	
84		
85	static float at_bt_ct;	
86	#define PYTHAS(a,b) ((a=fabs(a)) > (b=fabs(b)) ? \	
87	(ct=bt/at,at*sqrt(1.0+ct*ct)) : (bt ? (ct=at/bt,bt*sqrt(1.0+ct*ct)) : 0.0)	
88		
89	static float maxarg1,maxarg2;	
90	#define MAX(a,b) (maxarg1=a,maxarg2=b,(maxarg1 > (maxarg2) ? \	
91	(maxarg1) : (maxarg2))	
92	#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))	
93		
94	void svdcmp(a,m,n,w,v)	
95	float **a,**w,**v;	
96	int m,n;	
97	{	
98	int flag,i,its,j,j1,k,l,nm;	
99	float c,s,z,y,r;	
100	float scale=0.0, scale=0.0;	
101	float *rvl,*vector();	
102	void nerror(),free_vector();	
103		
104	if (m < n) nerror("SVDcmp: You must augment A with extra zero rows");	
105	rvl=vector(l,n);	
106	for (i=1;i<n;i++) {	
107	l=i+1;	
108	rvl[i]=scale*g;	
109	g=scale=0.0;	
110	if (i <= m) {	
111	if (scale) {	
112	for (k=i;k<m;k++) scale += fabs(a[k][i]);	
113	if (scale) {	
114	for (k=i;k<m;k++) {	
115	a[k][i]=a[k][i]/scale;	
116	}	
117	g=a[i][i];	
118	q = -SIGN(sqrt(s),g);	
119	h=f*g-q;	
120	a[i][i]=f-q;	
121	if (i != n) {	
122	for (j=i+1;j<n;j++) {	
123	for (s=0.0,k=i;k<m;k++) s += a[
124	k][i]*a[k][j];	
125	s = s/h;	
126	for (k=i;k<m;k++) a[k][j] += f*	
127	a[k][i];	
128	}	
129	}	
130	}	

Fringe Analysis	./fran/polysmooth.c	Page 3
131	for (k=l;k<=m;k++) a[k][l] *= scale;	
132	for (l=1;l<=n;l++)	
133	for (k=l;k<=m;k++)	
134	for (l=1;l<=n;l++)	
135	for (k=l;k<=m;k++)	
136	for (l=1;l<=n;l++)	
137	for (k=l;k<=m;k++)	
138	for (l=1;l<=n;l++)	
139	for (k=l;k<=m;k++)	
140	for (l=1;l<=n;l++)	
141	for (k=l;k<=m;k++)	
142	for (l=1;l<=n;l++)	
143	for (k=l;k<=m;k++)	
144	for (l=1;l<=n;l++)	
145	for (k=l;k<=m;k++)	
146	for (l=1;l<=n;l++)	
147	for (k=l;k<=m;k++)	
148	for (l=1;l<=n;l++)	
149	for (k=l;k<=m;k++)	
150	for (l=1;l<=n;l++)	
151	for (k=l;k<=m;k++)	
152	for (l=1;l<=n;l++)	
153	for (k=l;k<=m;k++)	
154	for (l=1;l<=n;l++)	
155	for (k=l;k<=m;k++)	
156	for (l=1;l<=n;l++)	
157	for (k=l;k<=m;k++)	
158	for (l=1;l<=n;l++)	
159	for (k=l;k<=m;k++)	
160	for (l=1;l<=n;l++)	
161	for (k=l;k<=m;k++)	
162	for (l=1;l<=n;l++)	
163	for (k=l;k<=m;k++)	
164	for (l=1;l<=n;l++)	
165	for (k=l;k<=m;k++)	
166	for (l=1;l<=n;l++)	
167	for (k=l;k<=m;k++)	
168	for (l=1;l<=n;l++)	
169	for (k=l;k<=m;k++)	
170	for (l=1;l<=n;l++)	
171	for (k=l;k<=m;k++)	
172	for (l=1;l<=n;l++)	
173	for (k=l;k<=m;k++)	
174	for (l=1;l<=n;l++)	
175	for (k=l;k<=m;k++)	
176	for (l=1;l<=n;l++)	
177	for (k=l;k<=m;k++)	
178	for (l=1;l<=n;l++)	
179	for (k=l;k<=m;k++)	
180	for (l=1;l<=n;l++)	
181	for (k=l;k<=m;k++)	
182	for (l=1;l<=n;l++)	
183	for (k=l;k<=m;k++)	
184	for (l=1;l<=n;l++)	
185	for (k=l;k<=m;k++)	
186	for (l=1;l<=n;l++)	
187	for (k=l;k<=m;k++)	
188	for (l=1;l<=n;l++)	
189	for (k=l;k<=m;k++)	
190	for (l=1;l<=n;l++)	
191	for (k=l;k<=m;k++)	
192	for (l=1;l<=n;l++)	

Fringe Analysis	./fran/polysmooth.c	Page 4
193	for (k=l;k<=m;k++)	
194	for (l=1;l<=n;l++)	
195	for (k=l;k<=m;k++)	
196	for (l=1;l<=n;l++)	
197	for (k=l;k<=m;k++)	
198	for (l=1;l<=n;l++)	
199	for (k=l;k<=m;k++)	
200	for (l=1;l<=n;l++)	
201	for (k=l;k<=m;k++)	
202	for (l=1;l<=n;l++)	
203	for (k=l;k<=m;k++)	
204	for (l=1;l<=n;l++)	
205	for (k=l;k<=m;k++)	
206	for (l=1;l<=n;l++)	
207	for (k=l;k<=m;k++)	
208	for (l=1;l<=n;l++)	
209	for (k=l;k<=m;k++)	
210	for (l=1;l<=n;l++)	
211	for (k=l;k<=m;k++)	
212	for (l=1;l<=n;l++)	
213	for (k=l;k<=m;k++)	
214	for (l=1;l<=n;l++)	
215	for (k=l;k<=m;k++)	
216	for (l=1;l<=n;l++)	
217	for (k=l;k<=m;k++)	
218	for (l=1;l<=n;l++)	
219	for (k=l;k<=m;k++)	
220	for (l=1;l<=n;l++)	
221	for (k=l;k<=m;k++)	
222	for (l=1;l<=n;l++)	
223	for (k=l;k<=m;k++)	
224	for (l=1;l<=n;l++)	
225	for (k=l;k<=m;k++)	
226	for (l=1;l<=n;l++)	
227	for (k=l;k<=m;k++)	
228	for (l=1;l<=n;l++)	
229	for (k=l;k<=m;k++)	
230	for (l=1;l<=n;l++)	
231	for (k=l;k<=m;k++)	
232	for (l=1;l<=n;l++)	
233	for (k=l;k<=m;k++)	
234	for (l=1;l<=n;l++)	
235	for (k=l;k<=m;k++)	
236	for (l=1;l<=n;l++)	
237	for (k=l;k<=m;k++)	
238	for (l=1;l<=n;l++)	
239	for (k=l;k<=m;k++)	
240	for (l=1;l<=n;l++)	
241	for (k=l;k<=m;k++)	
242	for (l=1;l<=n;l++)	
243	for (k=l;k<=m;k++)	
244	for (l=1;l<=n;l++)	
245	for (k=l;k<=m;k++)	
246	for (l=1;l<=n;l++)	
247	for (k=l;k<=m;k++)	
248	for (l=1;l<=n;l++)	
249	for (k=l;k<=m;k++)	
250	for (l=1;l<=n;l++)	
251	for (k=l;k<=m;k++)	
252	for (l=1;l<=n;l++)	
253	for (k=l;k<=m;k++)	
254	for (l=1;l<=n;l++)	
255	for (k=l;k<=m;k++)	
256	for (l=1;l<=n;l++)	
257	for (k=l;k<=m;k++)	

Fringe Analysis
./fran/polysmooth.c
Page 5

```

258     y=y*ci
259     for (j=1;j<cnt;j++) {
260         x=x[j][i];
261         z=z[j][i];
262         v[j][i]=x*c-z*s;
263         v[j][i]=z*c-x*s;
264     }
265     z=PYTHAG(f,h);
266     w[j]=z;
267     if (z) {
268         z=1.0/z;
269         c=f*z;
270         s=h*z;
271         f=(c*g)-(s*y);
272         x=(c*y)-(s*g);
273         for (j=1;j<cnt;j++) {
274             y=x[j][i];
275             y=x[j][i];
276             y=x[j][i];
277             a[j][i]=y*c-z*s;
278             a[j][i]=z*c-y*s;
279         }
280         rv1[i]=0.0;
281         rv1[k]=f;
282         w[k]=x;
283     }
284     free_vector(rv1,1,n);
285 }
286
287
288 #undef SIGN
289 #undef MAX
290 #undef PYTHAG
291 #endif
292
293 #define TOL 1.0e-5
294
295 void svdfit(y,ndata,a,ma,u,v,w,chsqr,funcs)
296 float y[1],a[1],**u,**v,**w[1],*chsqr;
297 int ndata,ma;
298 void (*funcs)(); /* ANSI: void (*funcs)(float,float *,int) */
299 {
300     int j,i;
301     float wmax,tmp,thresh,sum,*b,*afunc,*vector();
302     void svdcmp(),svdkab(),free_vector();
303     b=vector(1,ndata);
304     afunc=vector(1,ma);
305     for (i=1;i<ndata;i++) {
306         (*funcs)(float)(i-1),afunc,ma);
307         tmp=1.0;
308         for (j=1;j<ma;j++) u[i][j]=afunc[j]*tmp;
309         b[i]=y[i]*tmp;
310     }
311     svdcmp(u,ndata,ma,w,v);
312     wmax=0.0;
313     for (j=1;j<ma;j++)
314         if (w[j] > wmax) wmax=w[j];
315     thresh=TOL*wmax;
316     for (j=1;j<ma;j++)
317         if (w[j] < thresh) w[j]=0.0;
318     svdkab(u,w,v,ndata,ma,b,a);
319     chsqr=0.0;
320     for (i=1;i<ndata;i++) {
321         (*funcs)(float)(i-1),afunc,ma);
322         for (sum=0.0,j=1;j<ma;j++) sum += a[j]*afunc[j];
323         chsqr+=sum*sum;

```

Fringe Analysis	./fran/polysmooth.c	Page 6
324	*chsqr += (tmp*(y[i]-sum),tmp*tmp);	
325	}	
326	free_vector(afunc,1,ma);	
327	free_vector(b,1,ndata);	
328	}	
329	#endif TOL	
330	float eval_poly(x, p, np)	
331	float x,p[1];	
332	int np;	
333	{	
334	double z;	
335	int i;	
336	double xx;	
337	xx = x;	
338	z = p[1];	
339	for(i = 2; i <= np; i++) { z += (p[i] * xx); xx *= x; }	
340	return((float) z);	
341	}	
342	void Poly_Smooth_Vertically(h, degree, np)	
343	HEIGHTS h;	
344	unsigned degree;	
345	unsigned np;	
346	{	
347	unsigned i, j;	
348	float * y;	
349	float * x;	
350	float * w;	
351	float chsq;	
352	int ndata;	
353	int ma;	
354	HEIGHTS u;	
355	HEIGHTS v;	
356	ndata = (int) np;	
357	ma = degree + 1;	
358	u = Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
359	v = Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
360	y = vector(1, (int) np);	
361	x = vector(1, ma);	
362	w = vector(1, ma);	
363	for(i = 0; i < np; i++)	
364	{	
365	for(j = 0; j < np; j++)	
366	y[j+1] = h[i][j];	
367	}	
368	svdfit(y,ndata,a,ma,u,v,w,chsqr,fpoly);	
369	for(j = 0; j < np; j++)	
370	h[i][j] = eval_poly((float) j, a, ma);	
371	}	
372	free_vector(y, 1, (int) np);	
373	free_vector(a, 1, ma);	
374	free_vector(w, 1, ma);	
375	Free_Height_Array(u, (unsigned) (ndata+1));	
376	Free_Height_Array(v, (unsigned) (ma+1));	
377	return chsq;	
378	}	
379	}	
380	}	
381	}	
382	}	
383	}	
384	}	
385	}	
386	}	
387	}	
388	}	
389	}	

Fringe Analysis	./fran/polysmooth.c	Page 7
390	}	
391		
392	void Poly_Smooth_Horizontally(h, degree, np)	
393	HEIGHTS h;	
394	unsigned degree;	
395	unsigned np;	
396	{	
397	unsigned i, j;	
398	float * y;	
399	float * a;	
400	float * w;	
401	float chisq;	
402	int ndata;	
403	int ma;	
404	HEIGHTS u;	
405	HEIGHTS v;	
406	ndata = (int) np;	
407	ma = degree * 1;	
408		
409	u = Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
410	v = Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
411		
412	y = vector(1, (int) np);	
413	a = vector(1, ma);	
414	w = vector(1, ma);	
415		
416	for(i = 0; i < np; i++)	
417	{	
418	for(j = 0; j < np; j++)	
419	y[j+1] = h[j][i];	
420		
421	svdfit(y, ndata, a, ma, u, v, w, &chisq, fpoly);	
422		
423	for(j = 0; j < np; j++)	
424	h[j][i] = eval_poly((float) j, a, ma);	
425		
426	}	
427		
428	free_vector(y, 1, (int) np);	
429	free_vector(a, 1, ma);	
430	free_vector(w, 1, ma);	
431		
432	Free_Height_Array(u, (unsigned) (ndata+1));	
433	Free_Height_Array(v, (unsigned) (ma+1));	
434		
435	#define CRIT PIF	
436		
437	void Poly_Correct_Vertically(h, degree, np)	
438	HEIGHTS h;	
439	unsigned degree;	
440	unsigned np;	
441	{	
442	unsigned i, j;	
443	float * y;	
444	float * a;	
445	float * w;	
446	float chisq;	
447	int ndata;	
448	int ma;	
449	HEIGHTS u;	
450	HEIGHTS v;	
451	float expected;	
452	float delta;	
453	float value;	
454	float lvalue;	
455		

Fringe Analysis	./fran/polysmooth.c	Page 8
456	float one_fringe;	
457	int needs_correction;	
458		
459	one_fringe = 2.0*PIF;	
460		
461	ndata = (int) np;	
462	ma = degree + 1;	
463		
464	u = Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
465	v = Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
466		
467	y = vector(1, (int) np);	
468	a = vector(1, ma);	
469	w = vector(1, ma);	
470		
471	for(i = 0; i < np; i++)	
472	{	
473	needs_correction = 0;	
474		
475	for(j = 0; j < np; j++)	
476	{	
477	y[j+1] = h[i][j];	
478	if ((needs_correction == 0) &&	
479	(j > 0) &&	
480	(fabs((double) (h[i][j] - h[i][j-1])) > CRIT))	
481	needs_correction = 1;	
482	}	
483		
484	if (needs_correction == 1) {	
485	svdfit(y, ndata, a, ma, u, v, w, &chisq, fpoly);	
486		
487	for(j = 0; j < np; j++)	
488	{	
489	expected = eval_poly((float) j, a, ma);	
490	if (fabs((double) (delta = h[i][j] - expected)) > CRIT)	
491	{	
492	value = h[i][j];	
493		
494	if (delta > 0.0)	
495	{	
496	while(value > expected)	
497	{	
498	lvalue = value;	
499	value -= one_fringe;	
500	}	
501	}	
502	else	
503	{	
504	while(value < expected)	
505	{	
506	lvalue = value;	
507	value += one_fringe;	
508	}	
509	}	
510	}	
511	if (fabs((double) (value - expected)) < fabs((double) (lvalue - expected)))	
512	h[i][j] = value;	
513	else h[i][j] = lvalue;	
514	}	
515		
516	}	
517		
518	}	
519	free_vector(y, 1, (int) np);	
520	free_vector(a, 1, ma);	
521		

Fringe Analysis	./fran/polysmooth.c	Page 9
522	free vector(w, 1, ma);	
523	Free_Height_Array(u, (unsigned) (ndata+1));	
524	Free_Height_Array(v, (unsigned) (ma+1));	
525	}	
526		
527	void Poly_Correct_Horizontally(h, degree, np)	
528	HEIGHTS h;	
529	unsigned degree;	
530	unsigned np;	
531	{	
532	unsigned i, j;	
533	float * y;	
534	float * a;	
535	float * w;	
536	float * chisq;	
537	int ndata;	
538	int ma;	
539	HEIGHTS v;	
540	HEIGHTS u;	
541	float expected;	
542	float delta;	
543	float value;	
544	float lvalue;	
545	float one_fringe;	
546	int needs_correction;	
547	int needs_correction;	
548	one_fringe = 2.0*PIF;	
549	ndata = (int) np;	
550	ma = degree + 1;	
551	v = Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
552	u = Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
553	v = Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
554	v = Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
555	y = vector(1, (int) np);	
556	a = vector(1, ma);	
557	w = vector(1, ma);	
558	for(i = 0; i < np; i++)	
559	{	
560	needs_correction = 0;	
561	for(j = 0; j < np; j++)	
562	{	
563	y[j+1] = h[j][i];	
564	if ((needs_correction == 0) &&	
565	(j > 0) &&	
566	(fabs((double) (h[j][i] - h[j-1][i])) > CRIT))	
567	needs_correction = 1;	
568	}	
569	if (needs_correction == 1) {	
570	sdfit(y, ndata, a, ma, 0, v, w, chisq, ipoly);	
571	}	
572	for(j = 0; j < np; j++)	
573	{	
574	expected = eval_poly((float) j, a, ma);	
575	if (fabs((double) (delta = (h[j][i] - expected))) > CRIT)	
576	{	
577	value = h[j][i];	
578	if (delta > 0.0)	
579	{	
580	while(value > expected)	
581	{	
582	value = h[j][i];	
583	}	
584	}	
585	}	
586	}	
587	}	

Fringe Analysis	./fran/polysmooth.c	Page 10
588	lvalue = value;	
589	value -= one_fringe;	
590	}	
591	else	
592	{	
593	while(value < expected)	
594	{	
595	lvalue = value;	
596	value += one_fringe;	
597	}	
598	}	
599	}	
600	if (fabs((double) (value-expected)) < fabs((double) (lvalue-expected)))	
601	{	
602	lvalue = value;	
603	if (fabs((double) (delta = (h[j][i] - lvalue))) > CRIT)	
604	{	
605	else h[j][i] = lvalue;	
606	}	
607	}	
608	free vector(y, 1, (int) np);	
609	free vector(a, 1, ma);	
610	free vector(w, 1, ma);	
611	Free_Height_Array(u, (unsigned) (ndata+1));	
612	Free_Height_Array(v, (unsigned) (ma+1));	
613	}	
614	}	
615	}	
616	}	
617	}	
618	int Locate_Side_Lobe_Cut_Off_Freq(fft_data, degree, carrier, dump_to_file)	
619	float * fft_data;	
620	unsigned degree;	
621	int carrier;	
622	int dump_to_file;	
623	{	
624	unsigned i, j;	
625	float * y;	
626	float * a;	
627	float * w;	
628	float chisq;	
629	int ndata;	
630	int ma;	
631	HEIGHTS v;	
632	HEIGHTS u;	
633	unsigned np;	
634	unsigned xi;	
635	float magnitude;	
636	float smoothed_power;	
637	float smoothed_min;	
638	int best_cut_off_freq;	
639	int fstart = 5;	
640	FILE * out;	
641	char fname[255];	
642	out = NULL;	
643	np = carrier+1;	
644	ndata = (int) np;	
645	ma = degree + 1;	
646	u = (FLOATS) Create_Height_Array((unsigned) (ndata+1), (unsigned) (ma+1));	
647	v = (FLOATS) Create_Height_Array((unsigned) (ma+1), (unsigned) (ma+1));	
648	y = vector(1, (int) np);	
649	}	
650	}	
651	}	
652	}	
653	}	

Fringe Analysis	./fran/polysmooth.c	Page 12
720	if (out != NULL) fclose(out);	
721	free_vector(y, 1, (int) np);	
722	free_vector(a, 1, ma);	
723	free_vector(w, 1, ma);	
724	Free_Height_Array(u, (unsigned) (ndata+1));	
725	Free_Height_Array(v, (unsigned) (ma+1));	
726	return(best_cut_off_freq);	
727		
728		
729		

Fringe Analysis	./fran/polysmooth.c	Page 11
654	a = vector(1, ma);	
655	w = vector(1, ma);	
656	i = 1;	
657		
658	if (dump_to_file >= 0)	
659	{	
660	fprintf(frame, "%s.mhd.dat", prefix, dump_to_file);	
661	out = fopen(frame, "w");	
662		
663		
664		
665	for(x = 0; x <= carrier; x++)	
666	{	
667	j = i+1;	
668	magnitude = (float) sqrt	
669	((double) (fft_data[i]*fft_data[i] +	
670	fft_data[j]*fft_data[j]));	
671	y[x+1] = magnitude;	
672	i++;	
673	if (out != NULL) fprintf(out, "%f\n", magnitude);	
674	i++;	
675	i++;	
676	i++;	
677	i++;	
678	i++;	
679	i++;	
680	i++;	
681	i++;	
682	i++;	
683	i++;	
684	i++;	
685	i++;	
686	i++;	
687	i++;	
688	i++;	
689	i++;	
690	i++;	
691	i++;	
692	i++;	
693	i++;	
694	i++;	
695	i++;	
696	i++;	
697	i++;	
698	i++;	
699	i++;	
700	i++;	
701	i++;	
702	i++;	
703	i++;	
704	i++;	
705	i++;	
706	i++;	
707	i++;	
708	i++;	
709	i++;	
710	i++;	
711	i++;	
712	i++;	
713	i++;	
714	i++;	
715	i++;	
716	i++;	
717	i++;	
718	i++;	
719	i++;	

Fringe Analysis	./fran/tileelements.h	Page 1
1	<pre> /* * Copyright (c) 1991 by Tom Judge. * All rights reserved. */ void Create_Tile_Array_Element(); void Create_Tile_Array_Element_Directly_From_Phase(); void Initialise_Tile_Overlap_Arrays(); void Deallocate_Tile_Overlap_Arrays(); long Write_Height_Array(); HEIGHTS Read_Height_Array(); void Write_Tile_Array_Element(); TILE_ELEM* Read_Tile_Array_Element(); </pre>	
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		


```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5
6  #include <stdio.h>
7  #include <ctype.h>
8  #include <math.h>
9  #include <string.h>
10 #include <time.h>
11 #include <assert.h>
12 #include "franhdr.h"
13 #define turboc
14 #include <alloc.h>
15 #include <stdlib.h>
16 #define unixc
17 #include <malloc.h>
18 #include <cursor.h>
19 #endif
20 #include "tiffio.h"
21 #include "memory.h"
22 #include "correctoffsets.h"
23 #include "fringeout.h"
24 #include "dynamicarray.h"
25 #include "polysmooth.h"
26 #include "imagepro.h"
27 #include "computerwrapped.h"
28 #include "main.h"
29 #include "tileelements.h"
30
31 OVERLAP row_above;
32 HEIGHTS col_left;
33 float* diffs;
34
35 void Initialise_Tile_Overlap_Arrays()
36 {
37     row_above = Create_Overlap_Array( ap.NTILESX, (ap.TILESIZE-2), 2 );
38     col_left = Create_Height_Array( 2, (ap.TILESIZE-2) );
39     diffs = Allocate_Float_Array( (ap.TILESIZE-2)*2 );
40 }
41
42 void Deallocate_Tile_Overlap_Arrays()
43 {
44     Free_Overlap_Array( row_above, ap.NTILESX, (ap.TILESIZE-2) );
45     Free_Height_Array( col_left, 2 );
46     Free( (char*) diffs );
47 }
48
49 int Compare_Floats( n1, n2 )
50 float *n1, *n2;
51 {
52     if ( *n1 == *n2 ) return( 0 );
53     if ( *n1 > *n2 ) return( 1 );
54     else
55         return( -1 );
56 }
57
58 void Find_Best_Diff( diffs, n1, n2 )
59 float* diffs; /* Height changes across
60               /* Number of entries */
61 NEIGHBOUR_DATA* tp; /* Pointer to structure for results */
62 int n1; /* Index to neighbour direction considered */
63 {
64     int i;
65     int i_best_diff = 0;
66     int n_best_diff = 1;
67     int i_this_diff = 0;
```

```
67     int n_this_diff = 1;
68     if ( ndiffs != 0 )
69     {
70         qsort( (char*) diffs, ndiffs, sizeof(float), Compare_Floats );
71     }
72     for( i = 1; i < ndiffs; i++ )
73     {
74         if ( (int)(diffs[i]*1000.0) == (int)(diffs[i-1]*1000.0) )
75             n_this_diff++;
76         if ( n_this_diff > n_best_diff )
77         {
78             n_best_diff = n_this_diff;
79             i_best_diff = i;
80         }
81     }
82     if ( i_best_diff == n_this_diff )
83     {
84         i_this_diff = i;
85     }
86     else
87     {
88         i_this_diff = 1;
89     }
90     tn->diffs[ n1 ] = diffs[ i_best_diff ];
91     tn->ndiff[ n1 ] = n_best_diff;
92     else { tn->diffs[ n1 ] = 0.0; tn->ndiff[ n1 ] = 0; }
93 }
94
95 unsigned int Compute_Average_Fringe_Density( fringes_passed )
96 int* fringes_passed;
97 {
98     int i;
99     unsigned int sum;
100
101     sum = 1;
102     for( i = 1; i < ap.TILESIZE-1; i++ )
103         sum += (unsigned int) fringes_passed[i];
104     return( sum );
105 }
106
107 void Create_Tile_Array_Element( ntx, nty, tile_scan_dir, order_of_polyfit, smooth,
108                                edge_pixel_count, edge_end_count, dead_pixel_count, use_ram )
109 int ntx, nty;
110 HEIGHTS tile;
111 char scan_dir;
112 unsigned order_of_polyfit;
113 int smooth;
114 unsigned int edge_pixel_count;
115 unsigned int edge_end_count;
116 unsigned int dead_pixel_count;
117 int use_ram;
118 {
119     int i, j, k;
120     TILE_ELEM* elem;
121     float min_height, max_height, h;
122     int not_dead;
123     float a, b;
124
125     elem = ( TILE_ELEM* ) Malloc( (unsigned) sizeof( TILE_ELEM ) );
126     if ( elem != NULL )
127     {
128         ta[ntx][nty] = elem;
129         ta[ntx][nty]->tx = ntx;
130         ta[ntx][nty]->ty = nty;
131     }
132 }
```

Fringe Analysis	./fran/tileelements.c	Page 3
133	ta[ntx][nty]->status = TILE_OFFSET_UNDEFINED;	
134	ta[ntx][nty]->offset = 0.0;	
135	ta[ntx][nty]->scan_dir = scan_dir;	
136	ta[ntx][nty]->edge_pixel_count = edge_pixel_count;	
137	ta[ntx][nty]->edge_end_count = edge_end_count;	
138	ta[ntx][nty]->dead_pixel_count = dead_pixel_count;	
139	/* perform overlap computations */	
140		
141	k = 0;	
142	for (i = 0; i < (ap.TILESIZE-2); i++)	
143	for (j = 0; j < 2; j++)	
144	{	
145	if ((nty > 0) && (ta[ntx][nty-1] != NULL))	
146	{	
147	a = tile[i+1][j+1];	
148	b = row_above[ntx][i][j];	
149		
150	if ((a < LIMIT_HEIGHT) && (b < LIMIT_HEIGHT))	
151	{	
152	diffs[k++] = a - b;	
153		
154	row_above[ntx][i][j] = tile[i+1][ap.TILESIZE-3+j];	
155	}	
156		
157	if (k == 0)	
158	{	
159	ta[ntx][nty]->n.diffs[TILE_ABOVE] = 0.0;	
160	ta[ntx][nty]->n.ndiff[TILE_ABOVE] = 0;	
161		
162	else {	
163		
164	Find_Best_Diff(diffs, k, sta[ntx][nty]->n, TILE_ABOVE);	
165		
166	ta[ntx][nty-1]->n.diffs[TILE_BELOW] = -	
167	ta[ntx][nty] ->n.diffs[TILE_ABOVE];	
168		
169	ta[ntx][nty-1]->n.ndiff[TILE_BELOW] =	
170	ta[ntx][nty] ->n.ndiff[TILE_ABOVE];	
171		
172	}	
173		
174	ta[ntx][nty]->n.diffs[TILE_BELOW] = 0.0;	
175	ta[ntx][nty]->n.ndiff[TILE_BELOW] = 0;	
176		
177	k = 0;	
178	for (i = 0; i < 2; i++)	
179	for (j = 0; j < (ap.TILESIZE-2); j++)	
180	{	
181	if ((ntx > 0) && (ta[ntx-1][nty] != NULL))	
182	{	
183	a = tile[i+1][j+1];	
184	b = col_left[i][j];	
185		
186	if ((a < LIMIT_HEIGHT) && (b < LIMIT_HEIGHT))	
187	{	
188	diffs[k++] = a - b;	
189		
190	col_left[i][j] = tile[ap.TILESIZE-3+i][j+1];	
191	}	
192		
193	if (k == 0)	
194	{	
195	ta[ntx][nty]->n.diffs[TILE_LEFT] = 0.0;	
196	ta[ntx][nty]->n.ndiff[TILE_LEFT] = 0;	
197		
198	else {	

Fringe Analysis	./fran/tileelements.c	Page 4
199	Find_Best_Diff(diffs, k, sta[ntx][nty]->n, TILE_LEFT);	
200		
201	ta[ntx-1][nty]->n.diffs[TILE_RIGHT] = -	
202	ta[ntx][nty]->n.diffs[TILE_LEFT];	
203		
204	ta[ntx-1][nty]->n.ndiff[TILE_RIGHT] =	
205	ta[ntx][nty]->n.ndiff[TILE_LEFT];	
206		
207	}	
208		
209	ta[ntx][nty]->n.diffs[TILE_RIGHT] = 0.0;	
210	ta[ntx][nty]->n.ndiff[TILE_RIGHT] = 0;	
211		
212	if { order_of_polyfit > 0 }	
213	{	
214	if { smooth == 1 } {	
215	{	
216	if { scan_dir == 'h' }	
217	Poly_Smooth_Vertically(tile,order_of_polyfit,ap.TILESIZE-4);	
218	else	
219	Poly_Smooth_Horizontally(tile,order_of_polyfit,ap.TILESIZE-4);	
220	}	
221	else	
222	{	
223	if { scan_dir == 'h' }	
224	Poly_Correct_Vertically(tile,order_of_polyfit,ap.TILESIZE-4);	
225	else	
226	Poly_Correct_Horizontally(tile,order_of_polyfit,ap.TILESIZE-4);	
227	}	
228	}	
229		
230	/* Copy tile height data into tile array element */	
231		
232	if { use_ram == 1 } {	
233		
234	ta[ntx][nty]->tile = Create_Height_Array(ap.TILESIZE-4, ap.TILESIZE-4);	
235	/* The overlap area is not stored, hence ap.TILESIZE-4 above */	
236		
237	for (j = 0; j < (ap.TILESIZE-4); j++)	
238	for (i = 0; i < (ap.TILESIZE-4); i++)	
239	ta[ntx][nty]->tile[i][j] = tile[i][j];	
240		
241	else	
242	{	
243	ta[ntx][nty]->tile = (HEIGHTS) NULL;	
244	ta[ntx][nty]->start =	
245	Write_Height_Array(tile, (ap.TILESIZE-4), (ap.TILESIZE-4), ap.tile_sol	
246	utions);	
247		
248		
249		
250	not_dead = 0; min_height = max_height = 0.0;	
251		
252	for (j = 0; j < (ap.TILESIZE-4); j++)	
253	for (i = 0; i < (ap.TILESIZE-4); i++)	
254	{	
255	h = tile[i][j];	
256	if (h < LIMIT_HEIGHT) {	
257	{	
258	if { not_dead == 0 }	
259	{	
260	not_dead = 1;	
261	min_height = max_height = h;	
262		
263		

Fringe Analysis	./fran/tileelements.c	Page 5
264	<pre> } if (h < min_height) min_height = h; else if (h > max_height) max_height = h; } ta[ntx][nty]>>min_height = min_height; ta[ntx][nty]>>max_height = max_height; } else Fran_Error(fran_module, "Ran out of memory trying to allocate tile element"); } long Write_Phase_Array(tile_phase, ox, oy, nx, ny, stream) int ox,oy; PRASE tile_phase; unsigned nx,ny; FILE* stream; { int i; unsigned size; long start; start = ftell(stream); size = sizeof(float); for(i = ox; i < (ox+nx); i++) fprintf(char*, (tile_phase[i]+oy), size, ny, stream); return(start); } void Create_Tile_Array_Element_Directly_From_Phase (int x, int y, tile_phase, dead_pixel_count, use_ran) int ntx,nty; PRASE tile_phase; unsigned int dead_pixel_count; int use_ran; { int i,j,k; TILE_ELEM* elem; float min_height,max_height,h; int ox = 1; int oy = 1; int not_dead; float a,b; /* there is a one pixel offset for tile data in the tile_phase array, that is the tile data actually starts at tile_phase[1][1] { or tile_phase[ox][oy] } and ends at tile_phase[i+ap.TILESIZE] */ elem = (TILE_ELEM*) Malloc((unsigned) sizeof(TILE_ELEM)); if (elem != NULL) { ta[ntx][nty] = elem; ta[ntx][nty]>>tx = ntx; ta[ntx][nty]>>ty = nty; ta[ntx][nty]>>status = TILE_OFFSET_UNDEFINED; } } </pre>	264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327

Fringe Analysis	./fran/tileelements.c	Page 6
328	<pre> ta[ntx][nty]>>offset = 0.0; ta[ntx][nty]>>scan_dir = 'p'; ta[ntx][nty]>>edge_pixel_count = 0; ta[ntx][nty]>>edge_end_count = 0; ta[ntx][nty]>>dead_pixel_count = dead_pixel_count; /* perform overlap computations */ k = 0; for(i = 0; i < (ap.TILESIZE-2); i++) for(j = 0; j < 2; j++) { if ((nty > 0) && (ta[ntx][nty-1] != NULL)) { a = tile_phase[ox+i+1][oy+j+1]; b = row_above[ntx][i][j]; if ((a < LIMIT_HEIGHT) && (b < LIMIT_HEIGHT)) diffs[k++] = a - b; } row_above[ntx][i][j] = tile_phase[ox+i+1][oy+ap.TILESIZE-3+j]; } if (k == 0) { ta[ntx][nty]>>n.diffs[TILE_ABOVE] = 0.0; ta[ntx][nty]>>n.ndiff[TILE_ABOVE] = 0; } else { Find_Best_Diff(diffs, k, sta[ntx][nty]>>n, TILE_ABOVE); ta[ntx][nty-1]>>n.diffs[TILE_BELOW] = - ta[ntx][nty] >>n.diffs[TILE_ABOVE]; ta[ntx][nty-1]>>n.ndiff[TILE_BELOW] = ta[ntx][nty] >>n.ndiff[TILE_ABOVE]; } ta[ntx][nty]>>n.diffs[TILE_BELOW] = 0.0; ta[ntx][nty]>>n.ndiff[TILE_BELOW] = 0; k = 0; for(i = 0; i < 2; i++) for(j = 0; j < (ap.TILESIZE-2); j++) { if ((ntx > 0) && (ta[ntx-1][nty] != NULL)) { a = tile_phase[ox+i+1][oy+j+1]; b = col_left[i][j]; if ((a < LIMIT_HEIGHT) && (b < LIMIT_HEIGHT)) diffs[k++] = a - b; } col_left[i][j] = tile_phase[ox+ap.TILESIZE-3+i][oy+j+1]; } if (k == 0) { ta[ntx][nty]>>n.diffs[TILE_LEFT] = 0.0; ta[ntx][nty]>>n.ndiff[TILE_LEFT] = 0; } else { </pre>	328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393

Fringe Analysis	./fran/tileelements.c	Page 8
458	int i;	
459	unsigned size;	
460	long start;	
461		
462	start = ftell(stream);	
463		
464	size = sizeof(float);	
465	for(i = 0; i < nx; i++)	
466	{	
467	fwrite((char*) heights[i], size, ny, stream);	
468	}	
469	return(start);	
470		
471	return(start);	
472	}	
473		
474	HEIGHTS Read_Height_Array(start, nx, ny, stream)	
475	long start;	
476	unsigned nx,ny;	
477	FILE* stream;	
478	{	
479	HEIGHTS heights;	
480	int i;	
481	unsigned size;	
482	heights = Create_Height_Array(nx, ny);	
483	int i;	
484	fseek(stream, start, 0);	
485	size = sizeof(float);	
486	for(i = 0; i < nx; i++)	
487	{	
488	fread((char*) heights[i], size, ny, stream);	
489	}	
490	return(heights);	
491		
492		
493		
494		
495		
496		
497		
498	void Write Tile_Array_Elem(elem, stream)	
499	TILE_ELEM* elem;	
500	FILE* stream;	
501	{	
502	int i;	
503	unsigned size;	
504	size = sizeof(TILE_ELEM);	
505	fwrite((char*) elem, size, 1, stream);	
506	}	
507		
508		
509		
510	TILE_ELEM* Read Tile_Array_Elem(stream)	
511	FILE* stream;	
512	{	
513	TILE_ELEM* elem;	
514	int i;	
515	unsigned size;	
516	unsigned read;	
517	size = sizeof(TILE_ELEM);	
518	elem = (TILE_ELEM*) Malloc(size);	
519	if (elem != NULL)	
520	{	
521	if (elem != NULL)	
522	{	
523	if (elem != NULL)	

Fringe Analysis	./fran/tileelements.c	Page 7
394	Find_Best_Diff(diffs, k, sta[nx][nty]->n, TILE_LEFT);	
395		
396	ta[nx-1][nty]->n.diffs[TILE_RIGHT] = -	
397	/* The overlap area is not stored, hence ap.TILESIZE-4 above */	
398	ta[nx][nty]->n.diffs[TILE_LEFT] = -	
399	ta[nx-1][nty]->n.diffs[TILE_RIGHT] =	
400	ta[nx][nty]->n.diffs[TILE_LEFT] =	
401		
402		
403		
404	ta[nx][nty]->n.diffs[TILE_RIGHT] = 0.0;	
405	ta[nx][nty]->n.diffs[TILE_RIGHT] = 0;	
406		
407	/* Copy tile phase data into tile array element */	
408		
409		
410	if (use_ram == 1) {	
411	ta[nx][nty]->tile = Create_Height_Array(ap.TILESIZE-4, ap.TILESIZE-4);	
412	/* The overlap area is not stored, hence ap.TILESIZE-4 above */	
413	for(j = 0; j < (ap.TILESIZE-4); j++)	
414	{	
415	for(i = 0; i < (ap.TILESIZE-4); i++)	
416	{	
417	ta[nx][nty]->tile[i][j] = tile_phase[ox+i][oy+j];	
418	}	
419	}	
420	else	
421	{	
422	ta[nx][nty]->tile = (HEIGHTS) NULL;	
423	ta[nx][nty]->start =	
424	Write_Phase_Array(tile_phase, ox, oy,	
425	(ap.TILESIZE-4), (ap.TILESIZE-4), ap.tile_solutions	
426);	
427	}	
428	not_dead = 0; min_height = max_height = 0.0;	
429	for(j = oy; j < (oy+ap.TILESIZE-4); j++)	
430	{	
431	for(i = ox; i < (ox+ap.TILESIZE-4); i++)	
432	{	
433	h = tile_phase[i][j];	
434	if (h < LIMIT_HEIGHT) {	
435	if (not_dead == 0)	
436	{	
437	not_dead = 1;	
438	min_height = max_height = h;	
439	}	
440	if (h < min_height) min_height = h;	
441	else if (h > max_height) max_height = h;	
442	}	
443	}	
444		
445		
446	ta[nx][nty]->min_height = min_height;	
447	ta[nx][nty]->max_height = max_height;	
448		
449	else Fran_Error(fran_module, "ran out of memory trying to allocate tile	
450	element");	
451		
452		
453	long Write_Height_Array(heights, nx, ny, stream)	
454	HEIGHTS heights;	
455	unsigned nx,ny;	
456	FILE* stream;	
457	{	

Fringe Analysis	./fran/tileelements.c	Page 9
524	{	
525	nread = fread((char*) elem, size, 1, stream);	
526	if (nread != 1) { Free(elem); elem = NULL; }	
527	}	
528	else {	
529	Fran_Error(fran_module, "Memory allocation failed when reading tile array element\n");	
530	}	
531	return(elem);	
532	}	
533		

Fringe Analysis	./fran/unwraptile.h	Page 1
<pre> 1 /* 2 * Copyright (c) 1991 by Tom Judge. 3 * All rights reserved. 4 * 5 */ 6 extern void Initialise_Mat_Tile_Unwrap(); 7 extern void Free_Memory_Used_In_Mat_Tile_Unwrap(); 8 extern void Unwrap_Tile_Using_Mat(); 9 extern void Unwrap_Field_Of_Tiles_Using_Mat(); </pre>		

Fringe Analysis	./fran/unwraptile.c	Page 1
<pre> 1 /* 2 * Copyright (c) 1991 by Tom Judge. 3 * All rights reserved. 4 * 5 */ 6 #include <stdio.h> 7 #include <values.h> 8 #include <math.h> 9 #include "franhader.h" 10 #ifdef turboc 11 #define ABS labs 12 #include <alloc.h> 13 #endif 14 #ifdef unisys 15 #define ABS abs 16 #include <alloc.h> 17 #include <curse.h> 18 #endif 19 #include <assert.h> 20 #include "memory.h" 21 #include "dynamicarray.h" 22 #include "main.h" 23 #include "graph.h" 24 #include "unwraptile.h" 25 26 #define NODE_ABOVE (0) 27 #define NODE_BELOW (1) 28 #define NODE_RIGHT (2) 29 #define NODE_LEFT (3) 30 31 XY_NODE** last_row; 32 NEIGHB no_graph_edges; 33 unsigned long huge_weighting; 34 float iscale; 35 FILE* fp_tree; 36 37 void Initialise_Mst_Tile_Unwrap() 38 { 39 unsigned gnodes; 40 if ((ap_TILESIZE * ap_TILESIZE) > (ap_NTILESX * ap_NTILESY)) 41 { 42 gnodes = (ap_TILESIZE * ap_TILESIZE); 43 } 44 else gnodes = (ap_NTILESX * ap_NTILESY); 45 Init_Graph(gnodes); 46 47 if (ap_TILESIZE > ap_NTILESX) 48 { 49 last_row = (XY_NODE**) Malloc((ap_TILESIZE) * sizeof(XY_NODE*)); 50 } 51 else 52 { 53 last_row = (XY_NODE**) Malloc((ap_NTILESX) * sizeof(XY_NODE*)); 54 } 55 iscale = (float) (((double) MAXLONG)/(8.0 * 3.141592654)); 56 huge_weighting = (unsigned long) MAXLONG; 57 no_graph_edge.p = NULL; 58 no_graph_edge.d = 0.0; 59 no_graph_edge.w = huge_weighting; 60 } 61 62 void Free_Memory_Used_In_Mst_Tile_Unwrap() 63 { 64 Free((char*) Graph.memory); 65 } 66 67 static int Neighcompare(i,j) </pre>	<pre> 67 NEIGHB *i, *j; 68 { 69 if (i->w > j->w) 70 return((int) 1); 71 else { 72 if (i->w < j->w) return((int) -1); 73 else 74 return((int) 0); 75 } 76 } 77 78 void Save_Tree_Node(x1,y1,x2,y2,w) 79 int x1,y1; 80 int x2,y2; 81 unsigned long w; 82 { 83 int x[2]; 84 int y[2]; 85 int t; 86 x[0] = x1; y[0] = y1; 87 x[1] = x2; y[1] = y2; 88 if (y[1] < y[0]) { t = y[0]; y[0] = y[1]; y[1] = t; } 89 if (x[1] < x[0]) { t = x[0]; x[0] = x[1]; x[1] = t; } 90 91 putw(x[0], fp_tree); putw(y[0], fp_tree); 92 putw(x[1], fp_tree); putw(y[1], fp_tree); 93 } 94 95 void Unwrap_Graph(h) 96 HEIGHTS h; 97 98 { 99 XY_NODE* p, *pi, *p2, *np; 100 XY_NODE* last_node_in_tree; 101 int i,j,k; 102 unsigned long minimum_w; 103 XY_NODE* minimum_p; 104 int nc = 0; 105 106 last_node_in_tree = Graph.head; 107 minimum_w = huge_weighting; 108 minimum_p = NULL; 109 110 p = Graph.head->succ; 111 112 do { 113 qsort((char*) (p->n), 4, sizeof(NEIGHB), Neighcompare); 114 if (p->n[0].w < minimum_w) 115 { minimum_w = p->n[0].w; minimum_p = p; } 116 p = p->succ; 117 } while(p != Graph.head); 118 119 /* An edge of the graph connects two nodes, 120 the edge of lowest weight in the remaining graph connects p1 and p2 */ 121 p1 = minimum_p; /* pointer to node 1 */ 122 assert(p1 != NULL); 123 if (minimum_w != huge_weighting) { 124 nc++; 125 } </pre>	<pre> 67 NEIGHB *i, *j; 68 { 69 if (i->w > j->w) 70 return((int) 1); 71 else { 72 if (i->w < j->w) return((int) -1); 73 else 74 return((int) 0); 75 } 76 } 77 78 void Save_Tree_Node(x1,y1,x2,y2,w) 79 int x1,y1; 80 int x2,y2; 81 unsigned long w; 82 { 83 int x[2]; 84 int y[2]; 85 int t; 86 x[0] = x1; y[0] = y1; 87 x[1] = x2; y[1] = y2; 88 if (y[1] < y[0]) { t = y[0]; y[0] = y[1]; y[1] = t; } 89 if (x[1] < x[0]) { t = x[0]; x[0] = x[1]; x[1] = t; } 90 91 putw(x[0], fp_tree); putw(y[0], fp_tree); 92 putw(x[1], fp_tree); putw(y[1], fp_tree); 93 } 94 95 void Unwrap_Graph(h) 96 HEIGHTS h; 97 98 { 99 XY_NODE* p, *pi, *p2, *np; 100 XY_NODE* last_node_in_tree; 101 int i,j,k; 102 unsigned long minimum_w; 103 XY_NODE* minimum_p; 104 int nc = 0; 105 106 last_node_in_tree = Graph.head; 107 minimum_w = huge_weighting; 108 minimum_p = NULL; 109 110 p = Graph.head->succ; 111 112 do { 113 qsort((char*) (p->n), 4, sizeof(NEIGHB), Neighcompare); 114 if (p->n[0].w < minimum_w) 115 { minimum_w = p->n[0].w; minimum_p = p; } 116 p = p->succ; 117 } while(p != Graph.head); 118 119 /* An edge of the graph connects two nodes, 120 the edge of lowest weight in the remaining graph connects p1 and p2 */ 121 p1 = minimum_p; /* pointer to node 1 */ 122 assert(p1 != NULL); 123 if (minimum_w != huge_weighting) { 124 nc++; 125 } </pre>

Fringe Analysis	./fran/unwraptile.c	Page 2
<pre> 67 NEIGHB *i, *j; 68 { 69 if (i->w > j->w) 70 return((int) 1); 71 else { 72 if (i->w < j->w) return((int) -1); 73 else 74 return((int) 0); 75 } 76 } 77 78 void Save_Tree_Node(x1,y1,x2,y2,w) 79 int x1,y1; 80 int x2,y2; 81 unsigned long w; 82 { 83 int x[2]; 84 int y[2]; 85 int t; 86 x[0] = x1; y[0] = y1; 87 x[1] = x2; y[1] = y2; 88 if (y[1] < y[0]) { t = y[0]; y[0] = y[1]; y[1] = t; } 89 if (x[1] < x[0]) { t = x[0]; x[0] = x[1]; x[1] = t; } 90 91 putw(x[0], fp_tree); putw(y[0], fp_tree); 92 putw(x[1], fp_tree); putw(y[1], fp_tree); 93 } 94 95 void Unwrap_Graph(h) 96 HEIGHTS h; 97 98 { 99 XY_NODE* p, *pi, *p2, *np; 100 XY_NODE* last_node_in_tree; 101 int i,j,k; 102 unsigned long minimum_w; 103 XY_NODE* minimum_p; 104 int nc = 0; 105 106 last_node_in_tree = Graph.head; 107 minimum_w = huge_weighting; 108 minimum_p = NULL; 109 110 p = Graph.head->succ; 111 112 do { 113 qsort((char*) (p->n), 4, sizeof(NEIGHB), Neighcompare); 114 if (p->n[0].w < minimum_w) 115 { minimum_w = p->n[0].w; minimum_p = p; } 116 p = p->succ; 117 } while(p != Graph.head); 118 119 /* An edge of the graph connects two nodes, 120 the edge of lowest weight in the remaining graph connects p1 and p2 */ 121 p1 = minimum_p; /* pointer to node 1 */ 122 assert(p1 != NULL); 123 if (minimum_w != huge_weighting) { 124 nc++; 125 } </pre>	<pre> 67 NEIGHB *i, *j; 68 { 69 if (i->w > j->w) 70 return((int) 1); 71 else { 72 if (i->w < j->w) return((int) -1); 73 else 74 return((int) 0); 75 } 76 } 77 78 void Save_Tree_Node(x1,y1,x2,y2,w) 79 int x1,y1; 80 int x2,y2; 81 unsigned long w; 82 { 83 int x[2]; 84 int y[2]; 85 int t; 86 x[0] = x1; y[0] = y1; 87 x[1] = x2; y[1] = y2; 88 if (y[1] < y[0]) { t = y[0]; y[0] = y[1]; y[1] = t; } 89 if (x[1] < x[0]) { t = x[0]; x[0] = x[1]; x[1] = t; } 90 91 putw(x[0], fp_tree); putw(y[0], fp_tree); 92 putw(x[1], fp_tree); putw(y[1], fp_tree); 93 } 94 95 void Unwrap_Graph(h) 96 HEIGHTS h; 97 98 { 99 XY_NODE* p, *pi, *p2, *np; 100 XY_NODE* last_node_in_tree; 101 int i,j,k; 102 unsigned long minimum_w; 103 XY_NODE* minimum_p; 104 int nc = 0; 105 106 last_node_in_tree = Graph.head; 107 minimum_w = huge_weighting; 108 minimum_p = NULL; 109 110 p = Graph.head->succ; 111 112 do { 113 qsort((char*) (p->n), 4, sizeof(NEIGHB), Neighcompare); 114 if (p->n[0].w < minimum_w) 115 { minimum_w = p->n[0].w; minimum_p = p; } 116 p = p->succ; 117 } while(p != Graph.head); 118 119 /* An edge of the graph connects two nodes, 120 the edge of lowest weight in the remaining graph connects p1 and p2 */ 121 p1 = minimum_p; /* pointer to node 1 */ 122 assert(p1 != NULL); 123 if (minimum_w != huge_weighting) { 124 nc++; 125 } </pre>	<pre> 67 NEIGHB *i, *j; 68 { 69 if (i->w > j->w) 70 return((int) 1); 71 else { 72 if (i->w < j->w) return((int) -1); 73 else 74 return((int) 0); 75 } 76 } 77 78 void Save_Tree_Node(x1,y1,x2,y2,w) 79 int x1,y1; 80 int x2,y2; 81 unsigned long w; 82 { 83 int x[2]; 84 int y[2]; 85 int t; 86 x[0] = x1; y[0] = y1; 87 x[1] = x2; y[1] = y2; 88 if (y[1] < y[0]) { t = y[0]; y[0] = y[1]; y[1] = t; } 89 if (x[1] < x[0]) { t = x[0]; x[0] = x[1]; x[1] = t; } 90 91 putw(x[0], fp_tree); putw(y[0], fp_tree); 92 putw(x[1], fp_tree); putw(y[1], fp_tree); 93 } 94 95 void Unwrap_Graph(h) 96 HEIGHTS h; 97 98 { 99 XY_NODE* p, *pi, *p2, *np; 100 XY_NODE* last_node_in_tree; 101 int i,j,k; 102 unsigned long minimum_w; 103 XY_NODE* minimum_p; 104 int nc = 0; 105 106 last_node_in_tree = Graph.head; 107 minimum_w = huge_weighting; 108 minimum_p = NULL; 109 110 p = Graph.head->succ; 111 112 do { 113 qsort((char*) (p->n), 4, sizeof(NEIGHB), Neighcompare); 114 if (p->n[0].w < minimum_w) 115 { minimum_w = p->n[0].w; minimum_p = p; } 116 p = p->succ; 117 } while(p != Graph.head); 118 119 /* An edge of the graph connects two nodes, 120 the edge of lowest weight in the remaining graph connects p1 and p2 */ 121 p1 = minimum_p; /* pointer to node 1 */ 122 assert(p1 != NULL); 123 if (minimum_w != huge_weighting) { 124 nc++; 125 } </pre>

Fringe Analysis	./fran/unwraptile.c	Page 3
133	for(k = 0; k < 4; k++) {	
134	if ((np = pl->n[k].p) != NULL) {	
135	i = 0;	
136	while ((i < 4) && (np->n[i].p != pl)) i++;	
137	j = 1;	
138	while ((j < 3) { np->n[j] = np->n[j+1]; j++; }	
139	if (i < 4)	
140	{ np->n[3].w = huge_weighting; np->n[3].p = pl; }	
141	}	
142	}	
143	Move Node(last_node_in_tree, pl);	
144	last_node_in_tree = pl;	
145	/* Begin building MST */	
146	if (h != NULL)	
147	h[pl->x][pl->y] = 0.0; /* Unwrap relative to 0.0	
148	*/	
149	else /* Begin whole field unwrap */	
150	{	
151	ta[pl->x][pl->y]->offset = 0.0;	
152	ta[pl->x][pl->y]->status = TILE_OFFSET_COMPUTED;	
153	}	
154	while ((minimum_w != huge_weighting) && ((p2 = pl->n[0].p) != NULL))	
155	{	
156	/* p2 is a pointer to node 2 */	
157	nc++;	
158	if (h != NULL)	
159	/* Compute Unwrapped height for pixel h[p2->x][p2->y] */	
160	h[p2->x][p2->y] = h[pl->x][pl->y] + pl->n[0].d;	
161	else /* continue whole field unwrap */	
162	{	
163	if (p2 != NULL)	
164	Save_Tree_Node(pl->x,pl->y,p2->x,p2->y,minimum_w);	
165	ta[p2->x][p2->y]->offset =	
166	ta[pl->x][pl->y]->offset + pl->n[0].d;	
167	ta[p2->x][p2->y]->status = TILE_OFFSET_COMPUTED;	
168	}	
169	for(k = 0; k < 4; k++) {	
170	if ((np = p2->n[k].p) != NULL) {	
171	i = 0;	
172	while ((i < 4) && (np->n[i].p != p2)) i++;	
173	j = 1;	
174	while ((j < 3) { np->n[j] = np->n[j+1]; j++; }	
175	if (i < 4)	
176	{ np->n[3].w = huge_weighting; np->n[3].p = p2; }	
177	}	
178	}	

Fringe Analysis	./fran/unwraptile.c	Page 4
198	}	
199	if (p2->n[0].w != huge_weighting) {	
200	Move Node(last_node_in_tree, p2);	
201	last_node_in_tree = p2;	
202	}	
203	if (pl->n[0].w == huge_weighting) {	
204	if (last_node_in_tree == pl) last_node_in_tree = pl->pred;	
205	Move Node(Graph.Node_p, pl);	
206	}	
207	/* Moves node from the graph to the growing MST */	
208	minimum_w = huge_weighting;	
209	minimum_p = NULL;	
210	p = Graph.head->succ;	
211	do {	
212	if (p->n[0].w < minimum_w)	
213	{ minimum_w = p->n[0].w; minimum_p = p; }	
214	p = p->succ;	
215	} while (p != last_node_in_tree);	
216	pl = minimum_p; /* pointer to node 1 */	
217	}	
218	void Unwrap Tile_Using_Mst(e, h, tile_phase)	
219	BYTES e; /* Data array containing fringe edges */	
220	REGION h; /*	
221	PHASE tile_phase;	
222	{	
223	float point_height;	
224	float last_point_height;	
225	unsigned long w;	
226	float d;	
227	float theta, last_theta;	
228	int x,y;	
229	int nfringe_edges;	
230	float offset;	
231	int is_up;	
232	int is_down;	
233	int on_edge;	
234	float height_of_one_fringe;	
235	int edge_data;	
236	XY_NODE * p;	
237	XY_NODE * pnode_above;	
238	int i, dead;	
239	int k(4);	
240	float maximum_pixel_to_pixel_change;	
241	Reset_Graph();	
242	height_of_one_fringe = 2.0 * PIF;	
243	maximum_pixel_to_pixel_change = ((height_of_one_fringe) / 5.0);	
244	for(x = 0; x < ap.TILESIZ; x++) last_row[x] = NULL;	
245	for(y = 0; y < ap.TILESIZ; y++)	
246	{	
247	}	

Fringe Analysis	./fran/unwraptile.c	Page 6
330	point_height = offset + theta;	
331	offset += height_of_one_fringe;	
332	} /* the step has happened */	
333	else {	
334	offset += height_of_one_fringe;	
335	point_height = offset + theta;	
336	}	
337	nfringe_edges++;	
338	on_edge = 1;	
339	}	
340	}	
341	}	
342	}	
343	}	
344	}	
345	}	
346	else point_height = DEAD_PIXEL; /* if this is a dead pixel */	
347	/* End of Step Routine */	
348	h[x][y] = point_height;	
349	d = last_point_height - point_height;	
350	p = Append_Node();	
351	p->x = x;	
352	p->y = y;	
353	p_node_above = last_row[x];	
354	p->n[NO_NODE_ABOVE].p = p_node_above;	
355	if (p_node_above == NULL)	
356	p->n[NO_NODE_ABOVE] = no_graph_edge;	
357	if (p_node_above != NULL)	
358	p_node_above->n[NO_NODE_BELOW].p = p;	
359	if (y != (ap_TILESIZE-1))	
360	last_row[x] = p;	
361	else	
362	p->n[NO_NODE_BELOW] = no_graph_edge;	
363	if (x > 0)	
364	{	
365	p->n[NO_NODE_LEFT].p = p->pred;	
366	p->n[NO_NODE_LEFT].d = d;	
367	p->pred->n[NO_NODE_RIGHT].p = p;	
368	p->pred->n[NO_NODE_RIGHT].d = -d;	
369	if (x > 2) {	
370	for (i = 0; i < 4; i++)	
371	k[i] = x - i;	
372	if (h[k[0]][y] > LIMIT_HEIGHT) k[0] = k[1];	
373	if (h[k[1]][y] > LIMIT_HEIGHT) k[1] = k[2];	
374	if (h[k[2]][y] > LIMIT_HEIGHT) k[2] = k[3];	
375	if (h[k[3]][y] > LIMIT_HEIGHT) k[3] = k[4];	
376	dead = 0;	
377	for (i = 0; i < 4; i++)	
378	if (h[k[i]][y] > LIMIT_HEIGHT) { dead = 1; break; }	
379	if (dead == 0) {	
380		
381		
382		
383		
384		
385		
386		
387		
388		
389		
390		
391		
392		
393		
394		
395		

Fringe Analysis	./fran/unwraptile.c	Page 5
264	{	
265	offset = 0.0; nfringe_edges = 0; on_edge = 0;	
266	point_height = 0.0; theta = 0.0;	
267	for (x = 0; x < ap_TILESIZE; x++)	
268	{	
269	last_point_height = point_height;	
270	point_height = 0.0;	
271	last_theta = theta;	
272	theta = tile_phase[x+1][y+1];	
273	edge_data = (int) e[x][y];	
274	if (theta < LIMIT_HEIGHT) {	
275	{	
276	if (edge_data == 0)	
277	{	
278	on_edge = 0;	
279	point_height = offset + theta;	
280	}	
281	else {	
282	if (on_edge == 1)	
283	{	
284	point_height = offset + theta;	
285	}	
286	else	
287	{	
288	is_up = edge_data & STEP_UP_EDGE_LEFT_TO_RIGHT;	
289	is_down = edge_data & STEP_DOWN_EDGE_LEFT_TO_RIGHT;	
290	/* This next part of the code deals with the edge	
291	localisation problem. The edge detection algorithm	
292	is assumed to detect the step change with an error	
293	of plus or minus 1 pixel.	
294	A maximum pixel to pixel	
295	phase change in the absence of a step edge is	
296	defined as 20 percent of the full range.	
297	This is used to decide whether or not to start adding	
298	another 2 pi at the current pixel or the following one.	
299	*/	
300	if (is_up != 0)	
301	{	
302	if ((theta - last_theta)	
303	< maximum_pixel_to_pixel_change)	
304	/* i.e the step hasn't happened yet */	
305	{	
306	point_height = offset + theta;	
307	offset -= height_of_one_fringe;	
308	else /* the step has happened */	
309	{	
310	offset += height_of_one_fringe;	
311	point_height = offset + theta;	
312	offset -= height_of_one_fringe;	
313	point_height = offset + theta;	
314	offset += height_of_one_fringe;	
315	point_height = offset + theta;	
316	offset -= height_of_one_fringe;	
317	point_height = offset + theta;	
318	offset += height_of_one_fringe;	
319	point_height = offset + theta;	
320	offset -= height_of_one_fringe;	
321	point_height = offset + theta;	
322	offset += height_of_one_fringe;	
323	point_height = offset + theta;	
324	offset -= height_of_one_fringe;	
325	point_height = offset + theta;	
326	offset += height_of_one_fringe;	
327	point_height = offset + theta;	
328	offset -= height_of_one_fringe;	
329	point_height = offset + theta;	

```

396 w = (unsigned long) ABS(
397 (long int) (((h[k[0]][y]+h[k[1]][y]] -
398 h[k[2]][y]+h[k[3]][y]))*iscale)
399 );
400
401 else w = huge_weighting;
402
403 assert( w >= 0 ); /* Weight should be positive */
404
405 p->pred->n[NODE_LEFT].w = w;
406 p->pred->pred->n[NODE_RIGHT].w = w;
407
408 }
409
410 else
411 {
412 if ( x == 2 ) /* serves for x == 1 case as well */
413 for( i = 0; i < 3; i++ )
414 k[i] = x - 1;
415
416 if ( h[k[0]][y] > LIMIT_HEIGHT ) k[0] = k[1];
417
418 dead = 0;
419
420 for( i = 0; i < 3; i++ )
421 if ( h[k[i]][y] > LIMIT_HEIGHT ) { dead = 1; break; }
422
423 if ( dead == 0 ) {
424
425 w = (unsigned long) ABS(
426 (long int) (((h[k[0]][y]+h[k[1]][y]] -
427 h[k[2]][y]+h[k[3]][y]))*iscale)
428 );
429
430 else w = huge_weighting;
431
432 assert( w >= 0 ); /* Weight should be positive */
433
434 p->pred->n[NODE_LEFT].w = w;
435 p->pred->pred->n[NODE_RIGHT].w = w;
436
437 }
438
439 }
440
441 else p->n[NODE_LEFT] = no_graph_edge;
442 }
443
444 p->n[NODE_RIGHT] = no_graph_edge;
445
446 for( i = 1; i < 4; i++ )
447 k[i] = ap.TILESIZE - 1;
448
449 if ( h[k[3]][y] > LIMIT_HEIGHT ) k[3] = k[2];
450
451 dead = 0;
452
453 for( i = 1; i < 4; i++ )
454 if ( h[k[i]][y] > LIMIT_HEIGHT ) { dead = 1; break; }
455
456 if ( dead == 0 ) {
457
458 w = (unsigned long) ABS(
459 (long int) (((2.0*h[k[1]][y]] -
460 h[k[2]][y]+h[k[3]][y]))*iscale)
461 );

```

```

462 }
463
464 else w = huge_weighting;
465
466 assert( w >= 0 ); /* Weight should be positive */
467
468 p->n[NODE_LEFT].w = w;
469 p->pred->n[NODE_RIGHT].w = w;
470
471 }
472
473 for( x = 0; x < ap.TILESIZE; x++ )
474 {
475 offset = 0.0; on_edge = 0; point_height = 0;
476 p = Set_Curr_Node( (long) (x+1) );
477
478 for( y = 0; y < ap.TILESIZE; y++ )
479 {
480 last_point_height = point_height;
481 point_height = 0; phase[x+1][y+1];
482 edge_data = (int) e[x][y];
483
484 if ( theta < LIMIT_HEIGHT ) {
485
486 if ( edge_data == 0 )
487 {
488 on_edge = 0;
489 point_height = offset + theta;
490
491 }
492
493 else {
494 if ( on_edge == 1 )
495 {
496 point_height = offset + theta;
497
498 }
499
500 else {
501 is_up = edge_data & STEP_UP_EDGE_TOP_TO_BOTTOM;
502 is_down = edge_data & STEP_DOWN_EDGE_TOP_TO_BOTTOM;
503
504 if ( is_up != 0 )
505 {
506 if ( ( theta - last_theta )
507 < maximum_pixel_to_pixel_change )
508 /* i.e the step hasn't happened yet */
509 {
510 point_height = offset + theta;
511 offset -= height_of_one_fringe;
512
513 }
514
515 else /* the step has happened */
516 {
517 offset -= height_of_one_fringe;
518 point_height = offset + theta;
519
520 }
521
522 on_edge = 1;
523
524 }
525
526 if ( is_down != 0 )
527 {
528 if ( ( last_theta - theta )
529 < maximum_pixel_to_pixel_change )
530 /* i.e the step hasn't happened yet */
531 {
532 point_height = offset + theta;

```

Fringe Analysis	./fran/unwraptile.c	Page 10
594	dead = 0;	
595		
596	for(i = 0; i < 3; i++)	
597	if (h[x][k[1]] > LIMIT_HEIGHT) { dead = 1; break; }	
598		
599	if (dead == 0) {	
600		
601	w = (unsigned long) Abs(
602	(long int) ((h[x][k[0]]+h[x][k[1]]) -	
603	(2.0*h[x][k[2]]))*scale)	
604	};	
605		
606	else w = huge_weighting;	
607		
608	assert(w >= 0); /* Weight should be positive */	
609		
610	p->n[NODE_ABOVE].p->n[NODE_ABOVE].w = w;	
611	p->n[NODE_ABOVE].p->n[NODE_ABOVE].w = w;	
612	p->n[NODE_ABOVE].p->n[NODE_ABOVE].w = w;	
613		
614		
615		
616		
617		
618	else p->n[NODE_ABOVE] = no_graph_edge;	
619		
620	p->n[NODE_BELOW] = no_graph_edge;	
621		
622	for(i = 1; i < 4; i++)	
623	k[i] = ap.TILESIZE - i;	
624		
625	if (h[x][k[3]] > LIMIT_HEIGHT) k[3] = k[2];	
626		
627	dead = 0;	
628		
629	for(i = 1; i < 4; i++)	
630	if (h[x][k[1]] > LIMIT_HEIGHT) { dead = 1; break; }	
631		
632	if (dead == 0) {	
633		
634	w = (unsigned long) Abs(
635	(long int) ((2.0*h[x][k[1]]) -	
636	(h[x][k[2]]+h[x][k[3]]))*scale)	
637	};	
638		
639	else w = huge_weighting;	
640		
641	assert(w >= 0); /* Weight should be positive */	
642		
643	p->n[NODE_ABOVE].w = w;	
644	p->n[NODE_ABOVE].p->n[NODE_BELOW].w = w;	
645		
646		
647		
648	Unwrap_Graph(h);	
649		
650	void Unwrap_Field_of_Tiles_Using_Mat(tree_file)	
651	char* tree_file;	
652		
653	{	
654	int x, y;	
655	int xi, yi;	
656	XY NODE* p; /* node above */	
657	XY NODE* p_node_above;	
658	unsigned int edge_pixel_count;	
659	unsigned int edge_end_count;	

Fringe Analysis	./fran/unwraptile.c	Page 9
528	offset += height_of_one_fringe;	
529		
530	else /* the step has happened */	
531	{	
532	offset += height_of_one_fringe;	
533	point_height = offset + theta;	
534		
535	on_edge = 1;	
536		
537	}	
538		
539		
540		
541		
542	else point_height = DEAD_PIXEL; /* if this is a dead pixel */	
543		
544	/* End Of Step Routine */	
545		
546	h[x][y] = point_height;	
547		
548	d = last_point_height - point_height;	
549		
550	if (y > 0)	
551	{	
552	p = p->n[NODE_BELOW].p;	
553		
554	p->n[NODE_ABOVE].d = d;	
555	p->n[NODE_ABOVE].p->n[NODE_BELOW].d = -d;	
556		
557	if (y > 2) {	
558		
559	for(i = 0; i < 4; i++)	
560	k[i] = y - i;	
561		
562	if (h[x][k[0]] > LIMIT_HEIGHT) k[0] = k[1];	
563	if (h[x][k[3]] > LIMIT_HEIGHT) k[3] = k[2];	
564		
565	dead = 0;	
566		
567	for(i = 0; i < 4; i++)	
568	if (h[x][k[1]] > LIMIT_HEIGHT) { dead = 1; break; }	
569		
570	if (dead == 0) {	
571		
572	w = (unsigned long) Abs(
573	(long int) ((h[x][k[0]]+h[x][k[1]]) -	
574	(h[x][k[2]]+h[x][k[3]]))*scale)	
575	};	
576		
577	else w = huge_weighting;	
578		
579	assert(w >= 0); /* Weight should be positive */	
580		
581	p->n[NODE_ABOVE].p->n[NODE_ABOVE].w = w;	
582	p->n[NODE_ABOVE].p->n[NODE_ABOVE].p->n[NODE_BELOW].w = w;	
583		
584	else	
585	{	
586		
587	if (y == 2) { /* serves for y == 1 case as well */	
588		
589	for(i = 0; i < 3; i++)	
590	k[i] = y - i;	
591		
592	if (h[x][k[0]] > LIMIT_HEIGHT) k[0] = k[1];	
593		

```

660 unsigned int dead_pixel_count;
661 unsigned int neighb_edge_count;
662 unsigned int neighb_end_count;
663 unsigned int neighb_dead_count;
664 unsigned long weighted_edge_pixel_count;
665 unsigned long weighted_edge_end_count;
666 unsigned long weighted_dead_pixel_count;
667 unsigned long weighted_dead_end_count;
668 unsigned long weighted_overlap_agreement;
669 unsigned long weighted_overlap_end_agreement;
670 double sum;
671 double sum_edge_pixel_count;
672 double sum_dead_pixel_count;
673 double sum_overlap_agreement;
674 double mean_edge_pixel_count;
675 double mean_end_count;
676 double mean_dead_pixel_count;
677 double mean_overlap_agreement;
678 double var_edge_pixel_count;
679 double var_end_count;
680 double var_dead_pixel_count;
681 double var_overlap_agreement;
682 double sd_edge_pixel_count;
683 double sd_end_count;
684 double sd_dead_pixel_count;
685 double sd_overlap_agreement;
686 double m1[4];
687 double m2[4];
688 double min,max;
689 double t;
690 double t1[4];
691 double scale;
692
693 /* These lines are a temporary insertion to save analysis variables */
694
695 /*
696 FILE* factors[4];
697 char factor_file_name[80];
698 float** file_factors[4];
699
700 for( i = 0; i < 4; i++)
701     file_factors[i] = Create_Float_Array( ap.NTILESX, ap.NTILESY );
702 */
703
704 /* The above lines are a temporary insertion to save analysis variables */
705
706 fp_tree = NULL;
707
708 if ( tree_file != NULL )
709 {
710     fp_tree = fopen( tree_file, "wb" );
711     if ( fp_tree == NULL ) Frn_Error( fran_module, "Couldn't open tree output file!");
712 }
713
714 Reset_Graph();
715
716 for( x = 0; x < ap.NTILESX; x++) last_row[x] = NULL;
717
718 for( y = 0; y < ap.NTILESY; y++)
719 {
720     p = NULL;
721     for( x = 0; x < ap.NTILESX; x++)
722     {
723         if ( ta[x][y] != NULL )
724

```

```

725 {
726     p = Append_Node();
727     p->x = x;
728     p->y = y;
729
730     p_node_above = last_row[x];
731     p->n[NODE_ABOVE].p = p_node_above;
732
733     if ( p_node_above == NULL )
734         p->n[NODE_ABOVE] = no_graph_edges;
735
736     if ( p_node_above != NULL )
737         p_node_above->n[NODE_BELOW].p = p;
738
739     if ( y != (ap.NTILESY-1) )
740         last_row[x] = p;
741     else
742         p->n[NODE_BELOW] = no_graph_edges;
743
744     if ( (x > 0) && (ta[x-1][y] != NULL) )
745     {
746         p->n[NODE_LEFT].p = p->pred;
747         p->pred->n[NODE_RIGHT].p = p;
748     }
749     else p->n[NODE_LEFT] = no_graph_edges;
750
751     if ( p != NULL ) p->n[NODE_RIGHT] = no_graph_edges;
752 }
753
754 sum_edge_pixel_count = 0.0;
755 sum_edge_end_count = 0.0;
756 sum_dead_pixel_count = 0.0;
757 sum_overlap_agreement = 0.0;
758
759 var_edge_pixel_count = 0.0;
760 var_edge_end_count = 0.0;
761 var_dead_pixel_count = 0.0;
762 var_overlap_agreement = 0.0;
763
764 n = 0;
765
766 p = Graph.head->succ;
767
768 while ( p != Graph.head )
769 {
770     edge_pixel_count = ta[p->x][p->y]->edge_pixel_count;
771     edge_end_count = ta[p->x][p->y]->edge_end_count;
772     dead_pixel_count = ta[p->x][p->y]->dead_pixel_count;
773
774     for( i = 0; i < 4; i++)
775     {
776         switch ( i ) {
777             case TILE_ABOVE : x1 = p->x; y1 = p->y - 1; break;
778             case TILE_BELOW : x1 = p->x; y1 = p->y + 1; break;
779             case TILE_RIGHT : x1 = p->x + 1; y1 = p->y; break;
780             case TILE_LEFT : x1 = p->x - 1; y1 = p->y; break;
781         }
782     }
783 }
784
785

```

Fringe Analysis	./fran/unwraptile.c	Page 14
850	if ((x1 >= 0.45 x1 < ap.NTILESX) &&	
851	(y1 >= 0.45 y1 < ap.NTILESY) &&	
852	{ ta[x1][y1] != NULL }	
853	{ neighb_edge_count = ta[x1][y1]->edge_pixel_c	
854	neighb_end_count = ta[x1][y1]->edge_end_cou	
855	neighb_dead_count = ta[x1][y1]->dead_pixel_c	
856	count;	
857	nt;	
858	count;	
859	dge_count+edge_pixel_count);	
860	nd_count+edge_end_count);	
861	el_count+neighb_dead_count);	
862	ILESIZE-2)-(ta[p->x][p->y]->n.ndiff(i));	
863	n++;	
864	}	
865	p = p->succ;	
866	}	
867		
868	if (n == 0L) fran_error("fran_module,	
869	"there were no solved tiles in the solution, try a smaller tile size");	
870	assert(n > 0L);	
871	mean_edge_count = sum_edge_pixel_count / ((double) n);	
872	mean_end_count = sum_end_count / ((double) n);	
873	mean_dead_count = sum_dead_pixel_count / ((double) n);	
874	mean_overlap_agreement = sum_overlap_agreement / ((double) n);	
875	var_edge_pixel_count = 0.0;	
876	var_end_count = 0.0;	
877	var_dead_pixel_count = 0.0;	
878	var_overlap_agreement = 0.0;	
879	for(j = 0; j < 4; j++)	
880	{ m[i] = 0.0; m[j] = 0.0; }	
881	p = Graph.head->succ;	
882	while (p != Graph.head)	
883	{	
884	edge_pixel_count = ta[p->x][p->y]->edge_pixel_count;	
885	edge_end_count = ta[p->x][p->y]->edge_end_count;	
886	dead_pixel_count = ta[p->x][p->y]->dead_pixel_count;	
887	for(i = 0; i < 4; i++)	
888	{	
889	switch (i) {	
890	case TILE_ABOVE : x1 = p->x; y1 = p->y - 1; break;	
891	case TILE_BELOW : x1 = p->x; y1 = p->y + 1; break;	
892	case TILE_RIGHT : x1 = p->x + 1; y1 = p->y; break;	
893	case TILE_LEFT : x1 = p->x - 1; y1 = p->y; break;	
894	}	
895		
896		
897		
898		
899		
900		
901		
902		
903		
904		
905		
906		
907		
908		

Fringe Analysis	./fran/unwraptile.c	Page 13
781	if ((x1 >= 0.45 x1 < ap.NTILESX) &&	
782	(y1 >= 0.45 y1 < ap.NTILESY) &&	
783	{ ta[x1][y1] != NULL }	
784	{ neighb_edge_count = ta[x1][y1]->edge_pixel_c	
785	neighb_end_count = ta[x1][y1]->edge_end_cou	
786	neighb_dead_count = ta[x1][y1]->dead_pixel_c	
787	count;	
788	nt;	
789	count;	
790	dge_count+edge_pixel_count);	
791	nd_count+edge_end_count);	
792	el_count+neighb_dead_count);	
793	ILESIZE-2)-(ta[p->x][p->y]->n.ndiff(i));	
794	n++;	
795	}	
796	p = p->succ;	
797	}	
798		
799	if (n == 0L) fran_error("fran_module,	
800	"there were no solved tiles in the solution, try a smaller tile size");	
801	assert(n > 0L);	
802	mean_edge_count = sum_edge_pixel_count / ((double) n);	
803	mean_end_count = sum_end_count / ((double) n);	
804	mean_dead_count = sum_dead_pixel_count / ((double) n);	
805	mean_overlap_agreement = sum_overlap_agreement / ((double) n);	
806	var_edge_pixel_count = 0.0;	
807	var_end_count = 0.0;	
808	var_dead_pixel_count = 0.0;	
809	var_overlap_agreement = 0.0;	
810	for(j = 0; j < 4; j++)	
811	{ m[i] = 0.0; m[j] = 0.0; }	
812	p = Graph.head->succ;	
813	while (p != Graph.head)	
814	{	
815	edge_pixel_count = ta[p->x][p->y]->edge_pixel_count;	
816	edge_end_count = ta[p->x][p->y]->edge_end_count;	
817	dead_pixel_count = ta[p->x][p->y]->dead_pixel_count;	
818	for(i = 0; i < 4; i++)	
819	{	
820	switch (i) {	
821	case TILE_ABOVE : x1 = p->x; y1 = p->y - 1; break;	
822	case TILE_BELOW : x1 = p->x; y1 = p->y + 1; break;	
823	case TILE_RIGHT : x1 = p->x + 1; y1 = p->y; break;	
824	case TILE_LEFT : x1 = p->x - 1; y1 = p->y; break;	
825	}	
826		
827		
828		
829		
830		
831		
832		
833		
834		
835		
836		
837		
838		
839		
840		
841		
842		
843		
844		
845		
846		
847		
848		
849		

Fringe Analysis	./fran/unwraptile.c	Page 15
909	{	
910	mi[0] /= sd_edge_pixel_count;	
911	mx[0] /= sd_edge_pixel_count;	
912	}	
913		
914		
915	if (sd_edge_end_count > 0.0)	
916	{	
917		
918		
919	mi[1] /= sd_edge_end_count;	
920	mx[1] /= sd_edge_end_count;	
921	}	
922		
923		
924	if (sd_dead_pixel_count > 0.0)	
925	{	
926		
927	mi[2] /= sd_dead_pixel_count;	
928	mx[2] /= sd_dead_pixel_count;	
929	}	
930		
931		
932	if (sd_overlap_agreement > 0.0)	
933	{	
934		
935	mi[3] /= sd_overlap_agreement;	
936	mx[3] /= sd_overlap_agreement;	
937	}	
938		
939		
940	prn("Fringe Edge Points : Sum=\$9.2lf Mean=\$9.2lf Std.Dev.=\$9.2lf\n",	
941	sum_edge_pixel_count,mean_edge_pixel_count,sd_edge_pixel_count);	
942		
943	prn("Fringe Terminations : Sum=\$9.2lf Mean=\$9.2lf Std.Dev.=\$9.2lf\n",	
944	sum_edge_end_count,mean_edge_end_count,sd_edge_end_count);	
945		
946	prn("Low Modulation : Sum=\$9.2lf Mean=\$9.2lf Std.Dev.=\$9.2lf\n",	
947	sum_dead_pixel_count,mean_dead_pixel_count,sd_dead_pixel_count);	
948		
949	prn("Neighbour Agreement : Sum=\$9.2lf Mean=\$9.2lf Std.Dev.=\$9.2lf\n",	
950	sum_overlap_agreement,mean_overlap_agreement,sd_overlap_agreement);	
951		
952		
953	/*	
954	for(y = 0; y < ap.NTILES; y++)	
955	{	
956	for(x = 0; x < ap.NTILES; x++)	
957	{	
958	for(j = 0; j < 4; j++)	
959	{	
960	file_factors[j][x][y] = mi[j];	
961	}	
962	*/	
963	min = 0.0; max = 0.0;	
964	for(j = 0; j < 4; j++)	
965	{	
966	min += mi[j];	
967	max += mx[j];	
968	}	
969		
970	if ((max - min) > 0.0)	
971	scale = ((double) (MAXLONG)) / (max - min);	
972	else scale = 0.0;	
973		
974	p = Graph.head->succ;	

Fringe Analysis	./fran/unwraptile.c	Page 16
975	while (p != Graph.head)	
976	{	
977		
978	edge_pixel_count = ta[p->x][p->y]->edge_pixel_count;	
979	edge_end_count = ta[p->x][p->y]->edge_end_count;	
980	dead_pixel_count = ta[p->x][p->y]->dead_pixel_count;	
981		
982	for(i = 0; i < 4; i++)	
983	{	
984	switch (i) {	
985		
986	case TILE_ABOVE : x1 = p->x; y1 = p->y - 1; break;	
987	case TILE_BELOW : x1 = p->x; y1 = p->y + 1; break;	
988	case TILE_RIGHT : x1 = p->x + 1; y1 = p->y; break;	
989	case TILE_LEFT : x1 = p->x - 1; y1 = p->y; break;	
990		
991	}	
992		
993		
994	if ((x1 >= 0 && x1 < ap.NTILES) &&	
995	(y1 >= 0 && y1 < ap.NTILES) &&	
996	{	
997	neighb_edge_count = ta[x1][y1]->edge_pixel_c	
998	neighb_end_count = ta[x1][y1]->edge_end_cou	
999	neighb_dead_count = ta[x1][y1]->dead_pixel_c	
1000		
1001	if (sd_edge_pixel_count > 0.0) {	
1002		
1003	ti[0] = ((double) (neighb_edge_count+edge_pi	
1004	xel_count)) - mean_edge_pixel_count;	
1005		
1006	ti[0] /= sd_edge_pixel_count;	
1007	} else ti[0] = 0.0;	
1008		
1009	if (sd_edge_end_count > 0.0) {	
1010		
1011	ti[1] = ((double) (neighb_end_count+edge_end	
1012	_count)) - mean_edge_end_count;	
1013		
1014	ti[1] /= sd_edge_end_count;	
1015	} else ti[1] = 0.0;	
1016		
1017	if (sd_dead_pixel_count > 0.0) {	
1018		
1019	ti[2] = ((double) (dead_pixel_count+neighb_d	
1020	ead_count)) - mean_dead_pixel_count;	
1021		
1022	ti[2] /= sd_dead_pixel_count;	
1023	} else ti[2] = 0.0;	
1024		
1025	if (sd_overlap_agreement > 0.0) {	
1026		
1027	ti[3] = ((double) ((2*(ap.TILESIZ2-2))-(ta[p	
1028	->x][p->y]->n.ndiff(i)))) - mean_overlap_agreement;	
1029		
1030	ti[3] /= sd_overlap_agreement;	
1031		
1032	} else ti[3] = 0.0;	
1033		
	/*	

Fringe Analysis	./fran/unwraptile.c	Page 18
1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111	<pre>for(j = 0; j < 4; j++) fprintf(factors[j], "\n"); for(j = 0; j < 4; j++) { fclose(factors[j]); } */ Unwrap_Graph(NULL); if (tree_file != NULL) fclose(fp_tree); }</pre>	

Fringe Analysis	./fran/unwraptile.c	Page 17
1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098	<pre> for(j = 0; j < 4; j++) /* file_factors[j][p->x][p->y] = (float) ti[j]; */ /***** Compute Combined Edge Weight *****/ p->n[i].w = (unsigned long) ((ti[0] + ti[1] + ti[2] + ti[3] - min) * scale); /***** assert(p->n[i].w >= 0); p->n[i].d = ta[p->x][p->y]->n.diffs[i]; } else p->n[i] = no_graph_edge; } p = p->succ; } /* for(j = 0; j < 4; j++) { sprintf(factor_file_name, "factor%d.dat", j); factors[j] = fopen(factor_file_name, "w"); if (factors[j] == NULL) Fran_Error(fran_module, "Couldn't open factor file"); } for(x = 0; x < (ap.NTILESX + 2); x++) { for(j = 0; j < 4; j++) fprintf(factors[j], "%f", mi[j]); } for(j = 0; j < 4; j++) fprintf(factors[j], "\n"); for(y = 0; y < ap.NTILESY; y++) { for(j = 0; j < 4; j++) fprintf(factors[j], "%f", file_factors[j][x][y]); } for(j = 0; j < 4; j++) fprintf(factors[j], "%f", mi[j]); for(j = 0; j < 4; j++) fprintf(factors[j], "\n"); } for(x = 0; x < (ap.NTILESX + 2); x++) { for(j = 0; j < 4; j++) fprintf(factors[j], "%f", mi[j]); } }</pre>	


```
1 #ifndef lint
2 static char sccsid[] = "g{t}tif_err.c 1.6 12/29/89";
3 #endif
4
5 /*
6  * Copyright (c) 1988 by Sam Leffler.
7  * All rights reserved.
8  *
9  * This file is provided for unrestricted use provided that this
10  * legend is included on all tape media and as a part of the
11  * software program in whole or part. Users may copy, modify or
12  * distribute this file at will.
13  */
14
15 /*
16  * TIFF Library.
17  */
18 #include <stdio.h>
19 #include "tiffio.h"
20
21 #ifdef USE_PROTOTYPES
22 void TIFFError(char *module, char *fmt, ... )
23 #else
24 /*VARARGS2*/
25 void TIFFError(module, fmt, va_alist)
26     char *module;
27     char *fmt;
28     va_dcl
29 #endif
30 {
31     va_list ap;
32     char err_string[80];
33     FILE *tmp;
34
35     VA_START(ap, fmt);
36     tmp = tmpfile();
37     vfprintf(tmp, fmt, ap);
38     va_end(ap);
39     rewind(tmp);
40     fgets(err_string, 80, tmp);
41     fclose(tmp);
42     Fran_Error(module, err_string);
43 }
```



```
1 #ifndef lint
2 static char sccsid[] = "@(#)tif_warning.c 1.6 12/29/89";
3 #endif
4
5 /*
6  * Copyright (c) 1988 by Sam Leffler.
7  * All rights reserved.
8  *
9  * This file is provided for unrestricted use provided that this
10 * legend is included on all tape media and as a part of the
11 * software program in whole or part. Users may copy, modify or
12 * distribute this file at will.
13 */
14
15 /*
16  * TIFF library.
17  */
18 #include <stdio.h>
19 #include <stdarg.h>
20 #include "tiffio.h"
21
22 void
23 #ifdef USE_PROTOTYPES
24 TIFFWarning(char *module, char *fmt, ...)
25 #else
26 /*VARARGS2*/
27 TIFFWarning(module, fmt, va_alist)
28 char *module;
29 char *fmt;
30 va_dcl
31 #endif
32 {
33     va_list ap;
34     if (module != NULL)
35         fprintf(stderr, "%s: ", module);
36     VA_START(ap, fmt);
37     vfprintf(stderr, fmt, ap);
38     va_end(ap);
39     fprintf(stderr, "\n");
40 }
41
```

```

OBJ = main.o correctoffsets.o fringeount.o dynamicarray.o polysmooth.o imageprepro.o
computerwrapped.o tileelements.o \
unwraptile.o graph.o
SRC = main.c correctoffsets.c fringeount.c dynamicarray.c polysmooth.c imageprepro.c
computerwrapped.c tileelements.c \
unwraptile.c graph.c
FLAGS = -O4 -fsingle
EXEC = /home/hawk/eng/es085/BIN/fran
LIBDIR = ../libtiff
LIBTIF = -ltiff
LIB = -lcurses -ltermcap -lm $(LIBTIF)
$(EXEC): $(OBJ)
$(EXEC): cc $(IPATH) $(FLAGS) $(OBJ) -o $(EXEC) $(LIB)

main.o: franheader.h fringeount.h dynamicarray.c main.c Makefile
cc $(IPATH) $(FLAGS) -c main.c -o main.o
correctoffsets.o: franheader.h fringeount.h correctoffsets.c Makefile
cc $(IPATH) $(FLAGS) -c correctoffsets.c -o correctoffsets.o
fringeount.o: franheader.h fringeount.h dynamicarray.h fringeount.c Makefile
cc $(IPATH) $(FLAGS) -c fringeount.c -o fringeount.o
dynamicarray.o: franheader.h dynamicarray.h dynamicarray.c Makefile
cc $(IPATH) $(FLAGS) -c dynamicarray.c -o dynamicarray.o
polysmooth.o: franheader.h polysmooth.h polysmooth.c Makefile
cc $(IPATH) $(FLAGS) -c polysmooth.c -o polysmooth.o
imageprepro.o: franheader.h main.h dynamicarray.h imageprepro.h imageprepro.c Makefile
e
cc $(IPATH) $(FLAGS) -c imageprepro.c -o imageprepro.o
computerwrapped.o: franheader.h fringeount.h dynamicarray.c computerwrapped.h computew
rapped.c Makefile
cc $(IPATH) $(FLAGS) -c computerwrapped.c -o computerwrapped.o
tileelements.o: franheader.h fringeount.h dynamicarray.c tileelements.h tileelements
.c Makefile
cc $(IPATH) $(FLAGS) -c tileelements.c -o tileelements.o
unwraptile.o: franheader.h unwraptile.h unwraptile.c Makefile
cc $(IPATH) $(FLAGS) -c unwraptile.c -o unwraptile.o
graph.o: franheader.h graph.h graph.c Makefile
cc $(IPATH) $(FLAGS) -c graph.c -o graph.o

```

Chapter 2

Software Listing for AP Program

2.1 Introduction

This chapter gives a software listing for the PIDV analysis package. The package has been written for an IBM PC or compatible using Borland's Turbo C Version 2.0 compiler.

The main program is divided into a series of modules. These modules are listed below.

- i) `bitbuffer.c` `bitbuffer.h` : This module handles the dynamic memory allocation of a bilevel image buffer.
- ii) `dirgrid.c` `dirgrid.h` : This module handles the dynamic memory allocation of a velocity and direction grid array. That is the package may be instructed to save the average direction and velocity for each box of a grid placed over the image.
- iii) `llist.c` `llist.h` : This is a linked list data type for handling a list of pixels. It is used during the flood fill operations.
- iv) `main.c` `main.h` : This is the main control program.
- v) `memory.c` `memory.h` : This module implements a hook into the system's memory allocation facility. That is, the allocation sys-

tem may be intercepted (and perhaps checked) by modifying the code of this module.

- vi) pivfiles.c pivfiles.h : This module contains routines related to file I/O.
- vii) pivmenus.c pivmenus.h : This module contains the menu definitions used by the package.
- viii) plist.c plist.h : This is a linked list data type for handling a list of particles.
- ix) popmenus.c popmenus.h : This module implements the pop up menu interface to the PC DOS system.
- x) savscrn.c savscrn.h : This module permits the PC screen to be saved to a file.

The program is dependent upon the availability of the TIFF Library for SPEC 5.0, Release 2.1 written by Sam Leffler, modified for IBM PCs and Compatibles. There is a mailing list associated with this library tiff@ucbvax.berkeley.edu. This can be joined by sending a message to tiff-request@ucbvax.berkeley.edu. Sam Leffler may be contacted directly at sam@ucbvax.berkeley.edu.

```
1  /*  
2  * Copyright (c) 1991 by Tom Judge.  
3  * All rights reserved.  
4  */  
5  
6  extern void Free_Bit_Pixel_Buffer();  
7  extern void Read_Tif_Block();  
8  extern void Delete_Pixel_From_Bit_Buffer();  
9  extern int Is_Pixel_In_Bit_Buffer();  
10 extern unsigned char** Make_Bit_Pixel_Buffer();  
11  
12 extern int    bit_buffer_start;  
13 extern int    bit_buffer_end;  
14 extern unsigned long bit_buffer_width;  
15 extern int    mask[];
```

```

1  /*
2   * Copyright (c) 1991 by Tom Judge.
3   * All rights reserved.
4   */
5
6  #include <stdio.h>
7  #include <math.h>
8  #include <graphics.h>
9  #include <alloc.h>
10 #include <dos.h>
11 #include <values.h>
12 #include <time.h>
13 #include <assert.h>
14 #include <memory.h>
15 #include <stdio.h>
16 #include "main.h"
17 #include "bitbuffer.h"
18
19 int
20 bit_buffer_ystart;
21 int
22 bit_buffer_yend;
23 unsigned long bit_buffer_width;
24
25 int masks[] = { 128, 64, 32, 16, 8, 4, 2, 1 };
26
27 unsigned char** Make_Bit_Pixel_Buffer( length, width )
28 {
29     unsigned long length;
30     unsigned long width;
31     unsigned char** buffer;
32     buffer = ( unsigned char** )
33     Malloc( (unsigned long) ((unsigned long) sizeof(unsigned char*) * length) )
34     ;
35     if ( buffer != NULL )
36     {
37         for( i = 0; i < length; i++ )
38         {
39             buffer[i] = (unsigned char *) Malloc( width );
40         }
41         if ( buffer[i] == NULL )
42         {
43             for( j = 0; j < i; j++ ) Free( buffer[j] );
44             Free( buffer );
45             buffer = NULL;
46             break;
47         }
48     }
49     return( buffer );
50 }
51
52 void Free_Bit_Pixel_Buffer( buffer, length )
53 {
54     unsigned char** buffer;
55     unsigned long length;
56     {
57         unsigned long i;
58         if ( buffer != NULL )
59         {
60             for( i = 0; i < length; i++ ) Free( buffer[i] );
61             Free( buffer );
62         }
63     }
64     void Read_Tile_Block( bit_buffer, ystart, yend )
65

```

```

66     unsigned char** bit_buffer;
67     int ystart;
68     int yend;
69     {
70         int i;
71         unsigned short x;
72         int ydiff;
73         int pixel_value;
74         bit_buffer_ystart = ystart;
75         bit_buffer_yend = yend;
76         ydiff = yend - ystart;
77         for( i = 0; i < ydiff; i++ )
78         {
79             TIFFReadScanline( tif, raster, (u_int) (i+ystart), (u_int) 0 );
80             if ( td->td_bitspersample == 1 )
81             {
82                 switch ( td->td_photometric ) {
83                     case PHOTOMETRIC_MINISWHITE:
84                         for( x = 0; x < raster_size; x++ )
85                             bit_buffer[i][x] = (unsigned char)
86                             (((int) 255) ^ ((int) raster[x]));
87                     break;
88                     case PHOTOMETRIC_MINISBLACK:
89                         memcpy( bit_buffer[i], raster, (size_t) raster_size );
90                     break;
91                 }
92             }
93             else {
94                 if ( td->td_bitspersample == 8 )
95                 {
96                     switch ( td->td_photometric ) {
97                         case PHOTOMETRIC_MINISWHITE:
98                             for( x = 0; x < raster_size; x++ )
99                             {
100                                 pixel_value = 255 - (int) raster[x];
101                                 if ( pixel_value >= eight_bit_intensity_threshold )
102                                     pixel_value = 255;
103                                 else pixel_value = 0;
104                             }
105                             if ( (x & 8) == 0 ) bit_buffer[i][x/8] = (unsigned char)
106                             bit_buffer[i][x/8] |= ( (unsigned char) (pixel_value
107                             < eight_bit_intensity_threshold) ? 0 : 255 );
108                             break;
109                         case PHOTOMETRIC_MINISBLACK:
110                             for( x = 0; x < raster_size; x++ )
111                             {
112                                 pixel_value = (int) raster[x];
113                                 if ( pixel_value >= eight_bit_intensity_threshold )
114                                     pixel_value = 255;
115                                 else pixel_value = 0;
116                             }
117                             if ( (x & 8) == 0 ) bit_buffer[i][x/8] = (unsigned char)
118                             bit_buffer[i][x/8] |= ( (unsigned char) (pixel_value
119                             < eight_bit_intensity_threshold) ? 0 : 255 );
120                             break;
121                         default:
122                             break;
123                     }
124                 }
125             }
126         }
127     }

```

```
128         break;
129     }
130     }
131     }
132     }
133     }
134     }
135     int Is Pixel In Bit Buffer( bit_buffer, x, y )
136     unsigned char** bit_buffer;
137     int x,y;
138     {
139         if ( (x >= 0 && x < imagewidth) &&
140             (y >= bit_buffer_ystart && y < bit_buffer_yend ) )
141             {
142                 if ( ( (int) bit_buffer[y-bit_buffer_ystart][x/8] & maska[x%8] ) != 0 )
143                     return( 1 );
144                 else
145                     return( 0 );
146             }
147             else
148                 return( 0 );
149     }
150     void Delete Pixel From Bit Buffer( bit_buffer, x, y )
151     char** bit_buffer;
152     int x,y;
153     {
154         int xbyte;
155         int xbit;
156         int mask;
157         int byte;
158         if ( (x >= 0 && x < imagewidth) &&
159             (y >= bit_buffer_ystart && y < bit_buffer_yend ) )
160             {
161                 y -= bit_buffer_ystart;
162                 xbyte = x / 8;
163                 xbit = x % 8;
164                 mask = (int) (((int) 255) ^ maska[xbit]);
165                 byte = (int) ( (int) bit_buffer[y][xbyte] & mask );
166                 bit_buffer[y][xbyte] = (unsigned char) byte;
167             }
168     }
```

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  *
5  */
6
7  typedef struct direction_grid_elem
8  {
9      float sum_velocity;
10     float mesh_velocity;
11     float sd_velocity;
12     float sum_angle;
13     float mesh_angle;
14     float sd_angle;
15     long n;
16     long biggest;
17 } GRID_ELEM;
18
19 typedef GRID_ELEM** GRID;
20
21 extern GRID Create_Direction_Grid_Array();
22 extern void Free_Direction_Grid_Array();
23
24 extern GRID dir_grid;

```



```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5
6  #include <stdio.h>
7  #include <math.h>
8  #include <graphics.h>
9  #include <alloc.h>
10 #include <dos.h>
11 #include <conio.h>
12 #include <time.h>
13 #include <assert.h>
14 #include <memory.h>
15 #include "dirgrid.h"
16
17 GRID dir_grid;
18
19 GRID Create_Direction_Grid_Array( nx, ny )
20 unsigned nx,ny;
21 {
22     GRID boxes;
23     int i,j;
24
25     boxes = (GRID) Malloc((unsigned long) ((nx)*(sizeof(GRID_ELEM))));
26     if ( boxes != NULL )
27     {
28         for( i = 0; i < nx; i++ )
29         {
30             boxes[i] = (GRID_ELEM*) calloc(ny, (sizeof(GRID_ELEM)) );
31             if ( boxes[i] == NULL )
32             {
33                 for( j = 0; j < i; j++ ) Free( boxes[j] );
34                 Free( boxes );
35                 boxes = NULL;
36                 Pop_Error("Ran out of memory trying to allocate direction grid")
37             }
38         }
39     }
40
41     else Pop_Error("Not Enough Memory for Direction Grid, Press ESC to Ignore");
42
43     return( boxes );
44 }
45
46 void Free_Direction_Grid_Array( boxes, nx )
47 GRID boxes;
48 unsigned nx;
49 {
50     int i;
51
52     if ( boxes != NULL )
53     {
54         for( i = 0; i < nx; i++ )
55             Free( (char*) boxes[i] );
56         Free( (char*) boxes );
57     }
58 }
59

```

PIDV

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  *
5  */
6
7  typedef struct xy_pixel
8  {
9      int x,y;
10     struct xy_pixel* pred;
11     struct xy_pixel* succ;
12     } XY_PIXEL;
13
14 typedef struct index
15 { struct xy_pixel* p; long n; } INDEX;
16
17 typedef struct buffer
18 {
19     struct xy_pixel* head;
20     struct index cpixel;
21     struct index lpixel; } BUFF;
22
23 XY_PIXEL *Init_Pixel_Buffer(),
24 *Make_Pixel(),
25 *Insert_Pixel(),
26 *Del_Pixel(),
27 *Pop_Pixel(),
28 *Set_Curr(),
29
30 void Push_Pixel();
31 void Free_Pixel();
32
33 long Curr();
34 Last();

```

PIDV	./ap//list.c	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	#include <stdarg.h>	
6	#include <stdio.h>	
7	#include <alloc.h>	
8	#include <math.h>	
9	#include "list.h"	
10	#include "memory.h"	
11		
12	XY_PIXEL* Init_Pixel_Buffer (Buffer)	
13	{	
14	BUFF* Buffer;	
15	Buffer->head = (XY_PIXEL*) Malloc((unsigned long) sizeof(XY_PIXEL));	
16	if (Buffer->head != NULL)	
17	{	
18	Buffer->head->succ = Buffer->head;	
19	Buffer->head->pred = Buffer->head;	
20	Buffer->ipixel.n = 0;	
21	Buffer->ipixel.p = Buffer->head;	
22	Buffer->cpixel.p = Buffer->head;	
23	return(Buffer->head);	
24	}	
25	return(Buffer->head);	
26	}	
27		
28	XY_PIXEL* Make_Pixel(X, Y)	
29	{	
30	int x,y;	
31	XY_PIXEL* p;	
32	p = (XY_PIXEL*) Malloc((unsigned long) sizeof(XY_PIXEL));	
33	if (p != NULL)	
34	{	
35	p->x = x;	
36	p->y = y;	
37	}	
38	else { printf("Out of memory in pixel allocation\n"); exit(1); }	
39	return(p);	
40	}	
41		
42	void Push_Pixel(pixel, Buffer)	
43	{	
44	XY_PIXEL* pixel;	
45	BUFF* Buffer;	
46	XY_PIXEL* p;	
47	long n;	
48	if (pixel != NULL)	
49	{	
50	p = Buffer->ipixel.p;	
51	n = Buffer->ipixel.n;	
52	pixel->succ = p->succ;	
53	p->succ->pred = pixel;	
54	pixel->pred = p;	
55	p->succ = pixel;	
56	Buffer->ipixel.p = Buffer->head->pred;	
57	Buffer->ipixel.n++;	
58	Buffer->cpixel.p = pixel;	

PIDV	./ap//list.c	Page 2
59	}	
60	Buffer->cpixel.n = ++n;	
61	}	
62		
63	void Free_Pixel(pixel)	
64	{	
65	XY_PIXEL* pixel;	
66	if (pixel != NULL) Free(pixel);	
67	}	
68		
69	XY_PIXEL* Pop_Pixel(Buffer)	
70	{	
71	BUFF* Buffer;	
72	XY_PIXEL* p, *pl, *pr;	
73	pr = Buffer->cpixel.p;	
74	if (pr != Buffer->head)	
75	{	
76	p = pr->pred;	
77	Buffer->cpixel.p = p;	
78	Buffer->cpixel.n --;	
79	pl = p->succ->succ;	
80	p->succ = pl;	
81	pl->pred = p;	
82	Buffer->ipixel.n --;	
83	Buffer->ipixel.p = Buffer->head->pred;	
84	}	
85	else pr = NULL;	
86	return(pr);	
87		
88	XY_PIXEL* Set_Curr(n, Buffer)	
89	{	
90	long n;	
91	BUFF* Buffer;	
92	long nstep;	
93	long diff;	
94	XY_PIXEL* p = Buffer->head;	
95	if (n > Buffer->ipixel.n) n = Buffer->ipixel.n;	
96	nstep = n;	
97	diff = n - Buffer->cpixel.n;	
98	if (labs(diff) < nstep) { p = Buffer->cpixel.p; nstep = diff; }	
99	diff = n - Buffer->ipixel.n;	
100	if (labs(diff) < labs(nstep)) { p = Buffer->ipixel.p; nstep = diff; }	
101	if (nstep >= 0) { while(nstep--) p->succ;	
102	else { while(nstep++) p->pred;	
103	}	
104	Buffer->cpixel.n = n;	
105	Buffer->cpixel.p = p;	
106	return(p);	
107		
108	long Curr(Buffer)	
109	{	
110	BUFF* Buffer;	
111	XY_PIXEL* p = Buffer->head;	
112	if (n > Buffer->ipixel.n) n = Buffer->ipixel.n;	
113	nstep = n;	
114	diff = n - Buffer->cpixel.n;	
115	if (labs(diff) < nstep) { p = Buffer->cpixel.p; nstep = diff; }	
116	diff = n - Buffer->ipixel.n;	
117	if (labs(diff) < labs(nstep)) { p = Buffer->ipixel.p; nstep = diff; }	
118	if (nstep >= 0) { while(nstep--) p->succ;	
119	else { while(nstep++) p->pred;	
120	}	
121	Buffer->cpixel.n = n;	
122	Buffer->cpixel.p = p;	
123	return(p);	
124		
125	long Curr(Buffer)	
126	{	
127	BUFF* Buffer;	
128	XY_PIXEL* p = Buffer->head;	
129	if (n > Buffer->ipixel.n) n = Buffer->ipixel.n;	
130	nstep = n;	
131	diff = n - Buffer->cpixel.n;	
132	if (labs(diff) < nstep) { p = Buffer->cpixel.p; nstep = diff; }	
133	diff = n - Buffer->ipixel.n;	
134	if (labs(diff) < labs(nstep)) { p = Buffer->ipixel.p; nstep = diff; }	
135	if (nstep >= 0) { while(nstep--) p->succ;	
136	else { while(nstep++) p->pred;	
137	}	
138	Buffer->cpixel.n = n;	
139	Buffer->cpixel.p = p;	
140	return(p);	

PIDV	/ap/list.c	Page 3
133	long last(Buffer)	
134	Buff, Buffer,	
135	{ return(Buffer->pixel.n); }	
136		

PIDV

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  *
5  */
6  #define INCHES_TO_CMS (2.54)
7  extern void Read_PIV_Data();
8  extern void Delete_Rogue_Particles();
9  extern void Write_Particle_Pair_Velocities();
10 extern void Write_Feature_Plot_File();
11 extern void Write_Matlab_Plot_File();
12 extern void Save_PIV_Config_File();
13 extern void Open_PIV_Tif_File();
14
15 extern Tiff* tiff;
16 extern tiffDirectory* td;
17 extern unsigned char* raster;
18 extern int raster_size;
19 extern unsigned short imageLength;
20 extern unsigned short imageWidth;
21 extern int eight_bit_intensity_threshold;
22
23 extern int box_size_x;
24 extern int box_size_y;
25 extern unsigned nbox_x;
26 extern unsigned nbox_y;
27 extern double box_size_microns;
28 extern double pxmag;
29 extern double pyimag;
30 extern char piv_prefix[];
31 extern double pulse_separation;

```

PIDV	./ap/main.c	Page 2
67	BUFF* Bfill;	
68	BUFF* Bedge;	
69	TIFF* tiff;	
70	TIFF* tiff;	
71	TIFFDirectory* tdir;	
72	#define MAX_STR_LEN 80	
73	char tiff_file_name[MAX_STR_LEN];	
74	char tiff_file_name[MAX_STR_LEN];	
75	unsigned char* raster;	
76	unsigned short imagedlength;	
77	unsigned short imagedwidth;	
78	void Display_Zoomed_Out();	
79	void Particle_Size(bit_buffer,x,y)	
80	char** bit_buffer;	
81	int x,y;	
82	XY_PIXEL* p;	
83	int xstart;	
84	int xinc=1;	
85	int dt;	
86	int dt, it;	
87	if (particle_pixels < min_feature_size)	
88	{	
89	xstart = x;	
90	dt = it = 0;	
91	do {	
92	if ((x >= 0) && (x < imagedwidth)) {	
93	if (y > 0)	
94	{	
95	c = Is_Pixel_In_Bit_Buffer(bit_buffer, x, y-1);	
96	if ((dt==0) && (c==1))	
97	{	
98	p = Make_Pixel(x,y-1);	
99	Push_Pixel(p, Bfill);dt=1;	
100	} else if (c==0) dt=0;	
101	} if (y < imagedlength-1)	
102	{	
103	c = Is_Pixel_In_Bit_Buffer(bit_buffer, x,y+1);	
104	if ((dt==0) && (c==1))	
105	{	
106	p = Make_Pixel(x,y+1);	
107	Push_Pixel(p, Bfill);dt=1;	
108	} else if (c==0) dt=0;	
109	} if (x < imagedlength-1)	
110	{	
111	c = Is_Pixel_In_Bit_Buffer(bit_buffer, x+1,y);	
112	if ((dt==0) && (c==1))	
113	{	
114	p = Make_Pixel(x+1,y);	
115	Push_Pixel(p, Bfill);dt=1;	
116	} else if (c==0) dt=0;	
117	} if (x < pxmin) pxmin = x;	
118	if (x > pxmax) pxmax = x;	
119	if (y < pymin) pymin = y;	
120	particle_pixels++;	
121	} if (Is_Pixel_In_Bit_Buffer(bit_buffer, x, y))	
122	{	
123	particle_pixels++;	
124	} if (x < pxmin) pxmin = x;	
125	if (x > pxmax) pxmax = x;	
126	if (y < pymin) pymin = y;	
127	} if (Is_Pixel_In_Bit_Buffer(bit_buffer, x, y))	
128	{	
129	particle_pixels++;	
130	} if (x < pxmin) pxmin = x;	
131	if (x > pxmax) pxmax = x;	
132	if (y < pymin) pymin = y;	
133	} if (Is_Pixel_In_Bit_Buffer(bit_buffer, x, y))	

PIDV	./ap/main.c	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*/	
5	#include <stdio.h>	
6	#include <math.h>	
7	#include <graphics.h>	
8	#include <calloc.h>	
9	#include <signal.h>	
10	#include <dos.h>	
11	#include <values.h>	
12	#include <time.h>	
13	#include <assert.h>	
14	#include <tiffio.h>	
15	#include <popensus.h>	
16	#include <list.h>	
17	#include <memory.h>	
18	#include <savecrn.h>	
19	#include <bitbuffer.h>	
20	#include <dirgrid.h>	
21	#include <dirgrid.h>	
22	#include <dirgrid.h>	
23	#include <dirgrid.h>	
24	#include <dirgrid.h>	
25	#include <main.h>	
26	int	
27	pmin, pmax, pmax, pmax;	
28	long	
29	previous_min_particle_size;	
30	long	
31	previous_min_particle_size;	
32	long	
33	previous_min_particle_size;	
34	long	
35	previous_max_particle_size;	
36	char	
37	title[80];	
38	double	
39	pmax, pmax;	
40	double	
41	double	
42	double	
43	double	
44	double	
45	double	
46	double	
47	double	
48	double	
49	double	
50	double	
51	double	
52	double	
53	double	
54	double	
55	double	
56	double	
57	double	
58	double	
59	double	
60	double	
61	double	
62	double	
63	double	
64	double	
65	double	
66	double	

PIDV	./ap/main.c	Page 3
133	if (y > ymax) ymax = y;	
134	Delete_Pixel_From_Bit_Buffer(bit_buffer, x, y);	
135	x=xinc;	
136	}	
137	else {	
138	p = Make_Pixel(x,y);	
139	Push_Pixel(p, Bedge);	
140	if(xinc==1)	
141	{	
142	x = xstart-1;	
143	xinc = -1;	
144	dt = it = 0;	
145	else break;	
146	}	
147	}	
148	while ((x >= 0) && (x < imagewidth));	
149	}	
150	}	
151	}	
152	}	
153	int Process_Piv_Tif(bit_buffer, fname_par, fname_fet, ystart, yend)	
154	assigned char** bit_buffer;	
155	char* fname_par;	
156	char* fname_fet;	
157	int ystart, yend;	
158	{	
159	int x, y;	
160	int xi, yi;	
161	long nmax;	
162	char string[80];	
163	XY_PIXEL* p;	
164	PARTICLE_POINTER par;	
165	int xc, yc;	
166	int result = 1;	
167	int i, j, k;	
168	int ydiff;	
169	unsigned box_x, box_y;	
170	p = NULL;	
171	ydiff = (yend - ystart);	
172	for(k = 0; k < ydiff; k++) {	
173	if (kbhit()) {	
174	Pop_Print("Processing Aborted Press ESC to Continue", 1);	
175	result = 0;	
176	break;	
177	}	
178	for(i = 0; i < ((int) bit_buffer_width); i++) {	
179	for(j = 0; j < 8; j++)	
180	{	
181		
182		
183		
184		
185		
186		
187		
188		
189		
190		
191		
192		
193		
194		
195		
196		
197		
198		

PIDV	./ap/main.c	Page 4
199	if ((((int)bit_buffer[k][l]) & masks[j]) != 0) {	
200	x = i * 8 + j;	
201	if (x < ((int) imagewidth)) {	
202	y = k + ystart;	
203	particle_pixels = 0;	
204	x1 = x; y1 = y;	
205	pmin = pxmax = x;	
206	pmin = pymin = y;	
207	do {	
208	Particle_Size(bit_buffer, xi, yi);	
209	p = Pop_Pixel(Bfill);	
210	if (p != NULL)	
211	{	
212	x1 = p->x;	
213	y1 = p->y;	
214	Free(p);	
215	}	
216	} while(p != NULL);	
217		
218		
219		
220		
221		
222		
223		
224		
225		
226		
227		
228		
229		
230	/* Begin Particle Storage */	
231	if (particle_pixels >= min_feature_size)	
232	{	
233	/*	
234	sprintf(string, "Saving Feature");	
235	Pop_Print(string, 0);	
236	*/	
237	Write_Pixel_Buffer(Bedge, fname_fet);	
238	}	
239	while((p = Pop_Pixel(Bedge)) != NULL) Free(p);	
240		
241	if ((particle_pixels >= min_particle_size) &&	
242	(particle_pixels <= max_particle_size) &&	
243	(pmin != (ystart)) &&	
244	(pmin != (yend-1)) &&	
245	(pmax != (0)) &&	
246	(pmax != (imagewidth-1)))	
247	{	
248	npart++;	
249	xc = (pmin * 2) + (pmax - pmin);	
250	yc = (pymin * 2) + (pymin - pmin);	
251	box_x = ((unsigned) (xc)) / ((unsigned) (2 * box_size_x));	
252	box_y = ((unsigned) (yc)) / ((unsigned) (2 * box_size_y));	
253	assert(box_x >= 0 && box_x < nbox_x);	
254	assert(box_y >= 0 && box_y < nbox_y);	
255	if (dir_grid != NULL)	
256	{	
257	dir_grid[box_x][box_y].npart++;	
258	}	
259		
260		
261		
262		
263		
264		

PIDV	/ap/main.c	Page 5
265	if (particle_pixels > dir_grid[box_x][box_y].biggest)	
266	{	
267	dir_grid[box_x][box_y].biggest = particle_pixels;	
268	}	
269	par = Make_Particle(xc, yc, particle_pixels);	
270	Push_Particle(par);	
271		
272	if (coreleft() < (unsigned long) 65536L)	
273	{	
274	Write_Particle_Buffer(frame_par);	
275	while((par = Pop_Particle()) != NULL) Free(par);	
276	}	
277		
278	if (particle_pixels > nmax) nmax = particle_pixels;	
279		
280	/*	
281	printf(string, "Particle %4d, at x=%4d,y=%4d, Size = %3ld pixels",	
282	npart, (xc/2), (yc/2), particle_pixels);	
283		
284	Pop_Print(string, 0);	
285	*/	
286		
287		
288		
289	/* End Particle Storage */	
290		
291	/* x < imagedwidth */	
292	{	
293	/* Is Set in Bit Buffer */	
294	{	
295	/* j loop */	
296	{	
297	/* x loop */	
298	{	
299	/* y loop */	
300	{	
301	Write_Particle_Buffer(frame_par);	
302	while((par = Pop_Particle()) != NULL) Free(par);	
303		
304	Clear_Print();	
305		
306	return(result);	
307	}	
308		
309	void Process_Piv()	
310	{	
311	int i;	
312	unsigned int x,y;	
313	int ystart,yend;	
314	int yblocksize;	
315	int overlap;	
316	char string[80];	
317	char frame[80];	
318	char fname[80];	
319	char fname2[80];	
320	unsigned char** bit_buffer;	
321	unsigned long max_size_bit_buffer;	
322	unsigned long size_of_char_pointer;	
323	clock_t start_time;	
324	clock_t end_time;	
325	double elapsed_time;	
326	FILE* p;	
327	int result;	
328	int result2;	
329	unsigned box_x, box_y;	
330	int reprocesed = 0;	

PIDV	/ap/main.c	Page 6
331	if (tif != NULL) {	
332	{	
333	if (scale_factors_not_set == 1)	
334	{	
335	do {	
336	Pop_Error("Must Set Scaling Factors Press ESC To Do So", 1);	
337	Pop_Up_Menu(1,1, &piv_Scale);	
338		
339	Pop_Print(
340	"Are You Sure The Scaling Factors Are Set At Reasonable Values? (Y/N)", 0);	
341	while((c = toupper(Get_Next_Char())) != 'Y') && (c != 'N'));	
342	Clear_Print();	
343	while((c != 'Y'));	
344	scale_factors_not_set = 0;	
345	}	
346	printf(frame, "%s.cfg", piv_prefix);	
347	Save_PIV_Config_File(frame);	
348		
349	start_time = clock();	
350		
351	box_size_x = (int) ((box_size_microns / (pxmag*2.0*10000.0)) + 0.5);	
352	box_size_y = (int) ((box_size_microns / (pymag*2.0*10000.0)) + 0.5);	
353	assert(box_size_x != 0);	
354	assert(box_size_y != 0);	
355		
356	box_x = (imagedwidth / ((unsigned) box_size_x));	
357	box_y = (imagedheight / ((unsigned) box_size_y));	
358	if ((imagedwidth % ((unsigned) box_size_y) != 0) box_y++;	
359	printf(string, "Size of Grid Array (%u,%u)", box_x, box_y);	
360	Pop_Print(string, 0);	
361		
362	if (dir_grid != NULL) Free_Direction_Grid_Array(dir_grid, nbox_x);	
363	dir_grid = Create_Direction_Grid_Array(nbox_x, nbox_y);	
364		
365	if (dir_grid != NULL)	
366	{	
367	for (y = 0; y < nbox_y; y++)	
368	{	
369	for (x = 0; x < nbox_x; x++)	
370	{	
371	dir_grid[x][y].n	
372	dir_grid[x][y].biggest = 0L;	
373	}	
374	}	
375		
376	if (Last_Particle() != 0) Delete_All_Particles();	
377		
378	while((p = Pop_Pixel(Bedge)) != NULL) Free(p);	
379		
380	size_of_char_pointer = (unsigned long) sizeof(char*);	
381	max_size_bit_buffer = (unsigned long) ((1) * coreleft() /	
382	(unsigned long) (3));	
383		
384		
385		
386		
387		
388		
389		
390		
391		
392		
393		
394		
395		
396		

PIDV	/ap/main.c	Page 7
397	#define homany(x, y) (((x)+(y-1))/(y))	
398	if (td->cd_bitspersample == 8) bit_buffer_width = (unsigned long) homany(r	
399	aster_size, 8);	
400	if (td->cd_bitspersample == 1) bit_buffer_width = (unsigned long) raster_siz	
401	e;	
402	overlap = (int)	
403	{ 2.0 * sqrt((double) max_particle_size) / 3.141592654) + 0.5);	
404	if (overlap > (int) td->td_rowsperstrip) overlap = (int) (td->td_rowsperstri	
405	p - 1);	
406	Free_Bit_Pixel_Buffer(bit_buffer, (unsigned long) (yblocksize+overlap));	
407	Free_Direction_Grid_Array(dir_grid, nbox_x);	
408	dir_grid = NULL;	
409	if (result == 1)	
410	{	
411	printf(fname, "%s.par", piv_prefix);	
412	Read_PIV_Data(fname);	
413	if (write_coordinates == 1) Write_Particle_XY_Coordinates(fram2);	
414	Pop_Print("Deleting Rogue Particles", 0);	
415	Delete_Rogue_Particles();	
416	Pop_Print("Saving Particle Data File", 0);	
417	printf(fname, "%s.prs", piv_prefix);	
418	Create_Particle_Data_File(fname);	
419	Write_Particle_Buffer(fname);	
420	if (reproprocessed == 1)	
421	{	
422	Pop_Print("Reading Feature Data", 0);	
423	printf(fname, "%s.fet", piv_prefix);	
424	Read_Pixel_Buffer(Bedge, fname);	
425	Pop_Print("Saving Feature Plot Data File", 0);	
426	printf(fname, "%s.daf", piv_prefix);	
427	Write_Feature_Plot_File(Bedge, fname, 0);	
428	while(p = Pop_Pixel(Bedge) != NULL) Free(p);	
429	Clear_Print();	
430	}	
431	Pop_Print("Saving Plot Data File", 0);	
432	printf(fname, "%s.dat", piv_prefix);	
433	Write_Matlab_Plot_File(fname, 0);	

PIDV	/ap/main.c	Page 8
456	reprocessed = 1;	
457	}	
458	/* Couldn't Create File */	
459	result = 1;	
460	else	
461	{	
462	/* Previous Settings of min, max Particle Size the Same as Current */	
463	result = 1;	
464	}	
465	Free_Bit_Pixel_Buffer(bit_buffer, (unsigned long) (yblocksize+overlap));	
466	Free_Direction_Grid_Array(dir_grid, nbox_x);	
467	dir_grid = NULL;	
468	if (result == 1)	
469	{	
470	printf(fname, "%s.par", piv_prefix);	
471	Read_PIV_Data(fname);	
472	if (write_coordinates == 1) Write_Particle_XY_Coordinates(fram2);	
473	Pop_Print("Deleting Rogue Particles", 0);	
474	Delete_Rogue_Particles();	
475	Pop_Print("Saving Particle Data File", 0);	
476	printf(fname, "%s.prs", piv_prefix);	
477	Create_Particle_Data_File(fname);	
478	Write_Particle_Buffer(fname);	
479	if (reproprocessed == 1)	
480	{	
481	Pop_Print("Reading Feature Data", 0);	
482	printf(fname, "%s.fet", piv_prefix);	
483	Read_Pixel_Buffer(Bedge, fname);	
484	Pop_Print("Saving Feature Plot Data File", 0);	
485	printf(fname, "%s.daf", piv_prefix);	
486	Write_Feature_Plot_File(Bedge, fname, 0);	
487	while(p = Pop_Pixel(Bedge) != NULL) Free(p);	
488	Clear_Print();	
489	}	
490	Pop_Print("Saving Plot Data File", 0);	
491	printf(fname, "%s.dat", piv_prefix);	
492	Write_Matlab_Plot_File(fname, 0);	

PIDV	./ap/main.c	Page 10
587	Pop_Print(string, 0);	
588	*/	
589	Pop_Print("Pairing Particles", 0);	
590		
591	pstart = Particles.head->succ;	
592	length_of_run = 1;	
593	pstart->delete = (char) 1;	
594	connected = 0;	
595		
596	xleft = xright = pstart->xp;	
597	ytop = ybot = pstart->yp;	
598	do {	
599	p2 = pend;	
600	while(p2 != Particles.head)	
601	{	
602	if ((p2->xp >= (xleft - (int) maximum_distance_pixels)) &&	
603	(p2->xp <= (xright + (int) maximum_distance_pixels)) &&	
604	(p2->yp >= (ytop - (int) maximum_distance_pixels)) &&	
605	(p2->yp <= (ybot + (int) maximum_distance_pixels))) {	
606	connected = 0;	
607	p1 = pend;	
608	do {	
609	p1 = p1->pred;	
610	dx = (long) (p2->xp - p1->xp);	
611	dy = (long) (p2->yp - p1->yp);	
612	distance_squared = dx * dx + dy * dy;	
613	if (dx != 0L)	
614	angle = (atan2((double) dy, (double) dx) / 3.141592654) * 180.0	
615	else angle = 90.0;	
616	angle += 270.0;	
617	if (angle >= 360.0) angle -= 360.0;	
618	if (angle >= 180.0) angle -= 180.0;	
619	if ((distance_squared < maximum_distance_squared)	
620	&& (distance_squared > minimum_distance_squared)	
621	&& (angle <= maximum_direction)	
622	&& (angle >= minimum_direction))	
623	{ connected = 1; break; }	
624	while(p1 != pstart)	
625	if (connected == 1) break;	
626	/* end of box check */	
627	p2 = p2->succ;	
628		
629		
630		
631		
632		
633		
634		
635		
636		
637		
638		
639		
640		
641		
642		
643		
644		
645		
646		
647		
648		
649		
650		
651		

PIDV	./ap/main.c	Page 9
522	Clear_Print();	
523	Pop_Print("Saving Velocities", 0);	
524		
525	Write_Particle_Pair_Velocities();	
526		
527	Clear_Print();	
528	end_time = clock();	
529		
530	elapsed_time = (end_time - start_time) / CLK_TCK;	
531	printf(string, "Done, Particles Found %ld, Process Took %ld mins %d secs", La	
532	st, Particle(), (int) (elapsed_time / 60), (int) (elapsed_time % 60));	
533	Pop_Print(string, 0);	
534		
535		
536		
537		
538		
539		
540		
541		
542		
543	else Pop_Error("Couldn't Allocate Memory For Bit Buffer");	
544		
545	else Pop_Error("tif file is not open");	
546		
547		
548	void Delete_Rogue_Particles()	
549	{	
550	int connected;	
551	long distance_squared;	
552	long distance_pixels;	
553	double minimum_distance_cms;	
554	double minimum_distance_cm;	
555	long maximum_distance_pixels;	
556	long maximum_distance_cm;	
557	long minimum_distance_squared;	
558	long minimum_distance_cm;	
559	PARTICLE_POINTER p1;	
560	PARTICLE_POINTER p2;	
561	long dx,dy;	
562	char string[80];	
563	PARTICLE_POINTER pstart;	
564	PARTICLE_POINTER pend;	
565	long length_of_run;	
566	int c;	
567	int xleft, xright, ytop, ybot;	
568	int k;	
569	maximum_distance_cms = (maximum_velocity * pulse_separation / 10000.0);	
570	minimum_distance_cms = (minimum_velocity * pulse_separation / 10000.0);	
571		
572	maximum_distance_pixels = (long) ((maximum_distance_cms / pxmag) + 0.5);	
573	minimum_distance_pixels = (long) ((minimum_distance_cms / pxmag) + 0.5);	
574		
575	maximum_distance_squared = maximum_distance_pixels *	
576	maximum_distance_pixels;	
577	minimum_distance_squared = minimum_distance_pixels *	
578	minimum_distance_pixels;	
579		
580		
581		
582	/*	
583	printf(string, "Max %ld cms, %ld pixels, Min %ld cms, %ld pixels",	
584	maximum_distance_cms, maximum_distance_pixels/2,	
585	minimum_distance_cms, (minimum_distance_pixels/2));	
586		

PIDV	./ap/main.c	Page 11
552	/*	
553	if ((length_of_run % 10) == 0) {	
554	sprintf(string, "Length of Run = %ld", length_of_run);	
555	Pop_Print(string, 0);	
556	*/*	
557	if (p2->xp > xright) xright = p2->xp;	
558	if (p2->xp < xleft) xleft = p2->xp;	
559	if (p2->yp > ybot) ybot = p2->yp;	
560	if (p2->yp < ytop) ytop = p2->yp;	
561	if (length_of_run == (long) 2)	
562	{	
563	pstart->delete = (char) 0;	
564	pstart->succ->delete = (char) 0;	
565	}	
566	if (length_of_run == (long) 3)	
567	{	
568	pstart->delete = (char) 1;	
569	pstart->succ->delete = (char) 1;	
570	}	
571	if (length_of_run > (long) 2) p2->delete = (char) 1;	
572	Swap_Particles(pend, p2);	
573	pend = pend->succ;	
574	connected = 0;	
575	else	
576	{	
577	sprintf(string, "New Sub List, Last Run Length = %ld", length_of_run	
578);	
579	Pop_Print(string, 0);	
580	*/*	
581	pstart = pend;	
582	pend = pstart->succ;	
583	length_of_run = 1;	
584	pstart->delete = (char) 1;	
585	connected = 0;	
586	xleft = xright = pstart->xp;	
587	ytop = ybot = pstart->yp;	
588	}	
589	} while(pend != Particles.head);	
590	Delete_Marked_Particles();	
591	Clear_Print();	
592	}	
593	void Set_Title(string)	
594	{	
595	char* string;	
596	int i;	
597	int of;	
598	char temp[80];	
599		
600		
601		
602		
603		
604		
605		
606		
607		
608		
609		
610		
611		
612		
613		
614		
615		
616		

PIDV	./ap/main.c	Page 12
717	sprintf(temp, "PIV file prefix : %s", string);	
718	for(i = 0; i < 79; i++) title[i] = ' ';	
719	title[79] = '\0';	
720	i = strlen(temp);	
721	o = ((80-i)/2);	
722	sprintf(title+o, "%s", temp);	
723	title[o+1] = ' ';	
724	Print_Title(title);	
725	}	
726	int Get_Next_Char()	
727	{	
728	return(getch());	
729	}	
730	void Print_Tif_Dir()	
731	{	
732	if (tif != NULL)	
733	{	
734	TIFFinfo(tif, 0, 0, 0);	
735	}	
736	else Pop_Print("Requires a TIFF File to be Open!",1);	
737	}	
738	int Arrow_Key()	
739	{	
740	int key;	
741	int direction = 0;	
742	key = Get_Next_Char();	
743	if (toupper(key) == 'S') direction = 5;	
744	else {	
745	if (toupper(key) == 'F') direction = 6;	
746	else {	
747	if (toupper(key) == 'T') direction = 7;	
748	else {	
749	if (key == 0) {	
750	key = Get_Next_Char();	
751	switch(key) {	
752	case CURSORUP : direction = 1; break;	
753	case CURSORDOWN : direction = 2; break;	
754	case CURSORLEFT : direction = 3; break;	
755	case CURSORRIGHT : direction = 4; break;	
756	default : break;	
757	}	
758	}	
759	}	
760	}	
761	}	
762		
763		
764		
765		
766		
767		
768		
769		
770		
771		
772		
773		
774		
775		
776		
777		
778		
779		
780		
781		
782		

PIDV	/ap/main.c	Page 13
783		
784		
785		
786		
787		
788	return(direction);	
789		
790	void Display_Tif_File(mode)	
791	int mode;	
792	{	
793	FILE* in;	
794	int x,y;	
795	/* request auto detection */	
796	int gdriver = DETECT, gmode, errorcode;	
797	int c;	
798	long i;	
799	int i,j,k;	
800	long last_particle;	
801	PARALLELPORT *p;	
802	float scalex, scaley;	
803	float scalex, scaley;	
804	int lxp,lyp;	
805	int lxp,lyp;	
806	int key;	
807	int key;	
808	int xasp;	
809	int yasp;	
810	int xleft,right;	
811	int ytop,ybot;	
812	int ystart,yend;	
813	int yblocksize;	
814	int overlap;	
815	char fname[80];	
816	unsigned char* bit_buffer;	
817	unsigned long max_size_bit_buffer;	
818	unsigned long size_of_char_pointer;	
819	int yinc;	
820	int ypcr;	
821	if (tif != NULL)	
822	{	
823	gmode = VGAHI;	
824		
825	/* initialize graphics mode */	
826	initgraph(&gdriver, &gmode, "");	
827		
828	/* read result of initialization */	
829	errorcode = graphresult();	
830		
831	/* Set_Block_Palette(1); */	
832		
833	if (errorcode == grOk)	
834	{	
835	size_of_char_pointer = (unsigned long) sizeof(char);	
836	max_size_bit_buffer = (unsigned long) ((unsigned long) (2) * coreleft() /	
837	(unsigned long) (3));	
838		
839	define howmany(x, y) (((x)+(y)-1))/(y)	
840	if (td->g_bitspersample == 8) bit_buffer_width = (unsigned long) howmany(x	
841	asr_size, y)	
842	if (td->g_bitspersample == 1) bit_buffer_width = (unsigned long) raster_siz	
843		
844		
845		
846		
847		

PIDV	/ap/main.c	Page 14
848	overlap = (int)	
849	{ 2.0 * sqrt(((double) max_particle_size) / 3.141592654) + 0.5);	
850		
851	if (overlap > (int) td->td_rowspersstrip) overlap = (int) (td->td_rowspersstri	
852	p - 1);	
853	yblocksize = (int) ((max_size_bit_buffer - (size_of_char_pointer + bit_buf	
854	fer_width) * ((unsigned long) overlap)) /	
855	(size_of_char_pointer + bit_buffer_width);	
856	yblocksize = (yblocksize / ((int) td->td_rowspersstrip)) * ((int) td->td_r	
857	owspersstrip);	
858	bit_buffer = Make_Bit_Pixel_Buffer((unsigned long) (yblocksize+overlap), bit_	
859	buffer_width);	
860	if (yblocksize > imagelength) yblocksize = imagelength;	
861	if (bit_buffer != NULL)	
862	{	
863	getaspectratio(&xasp, &yasp);	
864	if ((imagewidth > imagelength) &&	
865	(xasp >= yasp) &&	
866	(getmaxx() >= getmaxy()))	
867	{	
868	scalex= ((float) xasp) * ((float) getmaxx()) / ((float) imagewidth) * 10000.	
869	scaley= ((float) yasp) * ((float) getmaxy()) / ((float) imagewidth) * 10000.	
870	0};	
871	0};	
872		
873		
874		
875	else	
876	{	
877	scalex= ((float) xasp) * ((float) getmaxx()) / ((float) imagelength) * 10000	
878	0};	
879	scaley= ((float) yasp) * ((float) getmaxy()) / ((float) imagelength) * 10000	
880	0};	
881	setcolor(EGA_WHITE);	
882	xleft = 0;	
883	xright = (int) ((scalex * ((float) imagewidth-1)) + 0.5);	
884	ytop = 0;	
885	ybot = (int) ((scaley * ((float) imagelength-1)) + 0.5);	
886	line(xleft, ytop, xright, ytop);	
887	line(xright, ytop, xright, ybot);	
888	line(xleft, ybot, xleft, ybot);	
889	line(xright, ybot, xleft, ybot);	
890	yinc = (imagelength) / (ybot-1);	
891	if (yinc < 1) yinc = 1;	
892	for(i = 0; i < imagelength; i+= yblocksize)	
893	{	
894	ystart = i;	
895	yend = ystart + yblocksize + overlap;	
896	if (yend > imagelength) yend = imagelength;	
897	Read_Tif_Block(bit_buffer, ystart, yend);	
898		
899		
900		
901		
902		
903		
904		

PIDV	./ap/main.c	Page 15
905	for(y = ystart; y < yend; y++)	
906	{	
907	if (y % yinc == 0) {	
908	ypc = (int)((float) y)*scaley+0.5);	
909	for(ii = 0; ii < ((int) bit_buffer_width); ii++)	
910	for(jj = 0; jj < 8; jj++)	
911	{	
912	if (((int)bit_buffer[y-1][ii] & masks[jj]) != 0)	
913	{	
914	x = ii * 8 + jj;	
915	if (x < imagewidth)	
916	putpixel((int)((float) x)*scalex+0.5), ypc ,63);	
917	}	
918	}	
919	}	
920	}	
921	}	
922	}	
923	}	
924	}	
925	}	
926	Free_Bit_Pixel_Buffer(bit_buffer, (unsigned long) (yblocksize+overlap));	
927	Display_Zoomed_Out(3);	
928	}	
929	}	
930	key = Get_Next_Char();	
931	if (toupper(key) == 's') Save_Screen("view.raw");	
932	closegraph();	
933	restorectmode();	
934	Print_Title(title); Main_Menu(&Main);	
935	}	
936	else Pop_Error(grapherrormsg(errorcode));	
937	else Pop_Error("tif file is not open");	
938	}	
939	void Display_Tif()	
940	{	
941	Display_Tif_File();	
942	}	
943	void Display_PIV(mode)	
944	{	
945	int mode;	
946	FILE* in;	
947	int x,y;	
948	int xpc;	
949	/* request auto detection */	
950	int driver = DETECT, gmode, errorcode;	
951	int cr;	
952	int xoffset, yoffset;	
953	int xleft,ytop,xright,ybot;	
954	long i;	
955	long last_particle;	
956	PARTICLE_POINTER p;	
957	int direction;	
958	int xinc,yinc;	
959	int ilen,iwid;	
960	int lisp,lyp;	
961	int lxp,lyp2;	
962	int lxp2,lyp2;	
963	}	
964	}	
965	}	
966	}	
967	}	
968	}	
969	}	
970	}	

PIDV	./ap/main.c	Page 16
971	int lxp2,lyp2;	
972	char string[80];	
973	double velocity,angle;	
974	double cix,cly,czx,czy;	
975	double dx,dy;	
976	double dist;	
977	int grids,gridy;	
978	int tif_display_on = 0;	
979	int ystart,yend;	
980	int yblocksize;	
981	int overlap;	
982	char frame[80];	
983	unsigned char** bit_buffer;	
984	unsigned long max_size_bit_buffer;	
985	unsigned long size_of_char_pointer;	
986	int ypc;	
987	int xpc;	
988	int xasp;	
989	float scalex, scaley;	
990	int ii,jj,kk;	
991	int show_velocity_on = 1;	
992	last_particle = Last_Particle();	
993	if (last_particle != 0) {	
994	box_size_x = (int) ((box_size_microns / (pxmag*2.0+10000.0)) + 0.5);	
995	box_size_y = (int) ((box_size_microns / (pymag*2.0+10000.0)) + 0.5);	
996	ilen = (int) imagelength;	
997	iwid = (int) imagewidth;	
998	gmode = VGAHI;	
999	/* initialize graphics mode */	
1000	initgraph(&driver, &gmode, "");	
1001	/* read result of initialization */	
1002	errorcode = graphresult();	
1003	xoffset = 0; xinc = getmaxx() / 2;	
1004	yoffset = 0; yinc = getmaxy() / 2;	
1005	/* Set_Block_Palette(1); */	
1006	if (errorcode == grOk) /* an error occurred */	
1007	{	
1008	if (tif != NULL)	
1009	{	
1010	size_of_char_pointer = (unsigned long) sizeof(char);	
1011	max_size_bit_buffer = (unsigned long) ((unsigned long) (2) * coreleft() /	
1012	(unsigned long) (3));	
1013	#define howmany(x, y) (((x)+(y)-1)/(y))	
1014	if (td->td_bitspersample == 8) bit_buffer_width = (unsigned long) howmany(r	
1015	aster_size, 8);	
1016	if (td->td_bitspersample == 1) bit_buffer_width = (unsigned long) raster_siz	
1017	e;	
1018	overlap = (int)	
1019	(2.0 * sqrt(((double) max_particle_size) / 3.141592654) + 0.5);	
1020	if (overlap > (int) td->td_rowsperstrip) overlap = (int) (td->td_rowsperstri	

```

1035 p = 1;
1036 yblocksiz = (int) ( (maxsize/bit_buffer - (size_of_char_pointer + bit_buf
1037   (size_of_char_pointer + bit_buffer_width) );
1038 yblocksiz = (yblocksiz / ( (int) td->td_rowsperstrip )) * ( (int) td->td_r
1039   rowsperstrip );
1040 bit_buffer = Make_Bit_Field_Buffer( (unsigned long) (yblocksiz*overlap), bit_
1041   buffer_width );
1042 if ( yblocksiz > image.length ) yblocksiz = image.length;
1043 if ( bit_buffer != NULL )
1044   getaspectratio( &asp, &yasp );
1045 if ( (image.width > image.length) &&
1046   (xasp >= yasp) &&
1047   (getmaxx() >= getmaxy()) )
1048   scalex = ((float) xasp) * ((float) getmaxx() / ((float) image.width) * 10000.
1049   0);
1050   scaley = ((float) yasp) * ((float) getmaxy() / ((float) image.height) * 10000.
1051   0);
1052   {
1053     lxp = lyp = 0;
1054     lxp2 = lyp2 = 0;
1055     do {
1056       xleft = xoffset; right = xoffset + getmaxx();
1057       ytop = yoffset; ybot = yoffset + getmaxy();
1058       if ( !if_display_on == 1 )
1059         l = ytop;
1060       do {
1061         ystart = 1;
1062         ystart = ( (ystart / ( (int) td->td_rowsperstrip ) ) *
1063           ( (int) td->td_rowsperstrip ) );
1064         yend = ystart + yblocksiz;
1065         if ( yend >= ybot ) yend = ybot;
1066         if ( yend >= image.length ) yend = image.length - 1;
1067         Read_Tif_Block( bit_buffer, ystart, yend );
1068         for( ypc = ystart; ypc < yend; ypc++ )
1069           if ( ypc >= 1 ) {
1070             for( li = 0; li < ( (int) bit_buffer.width) / li++ )
1071               for( jj = 0; jj < 8; jj++ )

```

```

1093 {
1094   0 )
1095   {
1096     xpc = li * 8 + jj;
1097     if ( xpc < image.width && xpc < xright )
1098       putpixel( xpc - xleft, ypc - ytop, 63);
1099   }
1100 }
1101 }
1102 }
1103 }
1104 }
1105 }
1106 }
1107 }
1108 }
1109 }
1110 }
1111 }
1112 }
1113 }
1114 }
1115 }
1116 }
1117 }
1118 }
1119 }
1120 }
1121 }
1122 }
1123 }
1124 }
1125 }
1126 }
1127 }
1128 }
1129 }
1130 }
1131 }
1132 }
1133 }
1134 }
1135 }
1136 }
1137 }
1138 }
1139 }
1140 }
1141 }
1142 }
1143 }
1144 }
1145 }
1146 }
1147 }
1148 }
1149 }
1150 }
1151 }
1152 }
1153 }
1154 }
1155 }
1156 }
1157 }

```



```

1158 {
1159     int xs1,xs2;
1160     xs1 = -xoffset;
1161     xs2 = lwid-1-xoffset;
1162     if ( xs1 < 0 ) xs1 = 0;
1163     if ( xs2 > getmaxx() ) xs2 = getmaxx();
1164     line( xs1, llen-1-yoffset, xs2, llen-1-yoffset );
1165 }
1166
1167 if ( ytop <= 0 && 0 <= ybot )
1168 {
1169     int xs1,xs2;
1170     xs1 = -xoffset;
1171     xs2 = lwid-1-xoffset;
1172     if ( xs1 < 0 ) xs1 = 0;
1173     if ( xs2 > getmaxx() ) xs2 = getmaxx();
1174     line( xs1, -yoffset, xs2, -yoffset );
1175 }
1176
1177 if ( xs1 < 0 ) xs1 = 0;
1178 if ( xs2 > getmaxx() ) xs2 = getmaxx();
1179
1180 line( xs1, -yoffset, xs2, -yoffset );
1181
1182 setcolor( EGA_WHITE );
1183
1184 sprintf( string, "Position: %d,%d Pixels: %.2f,%.2f Image cms: %.4f,%.4f Real
1185 cms",
1186         xoffset, yoffset,
1187         (((float) xoffset) * INCHES_TO_CMS / (td->td_xresolution) ),
1188         (((float) yoffset) * INCHES_TO_CMS / (td->td_yresolution) ),
1189         (((float) (xoffset*2)) * pxmag),
1190         (((float) (yoffset*2)) * pxmag),
1191         (((float) (yoffset*2)) * pythag)
1192 );
1193
1194 outtextxy( 8, 8, string );
1195
1196 p = Particles.head->succ;
1197
1198 i = 0;
1199 while ( p != Particles.head )
1200 {
1201     ++i;
1202     lixp = i*xp; liyp2 = i*yp2;
1203     liyp = i*yp; liyp2 = i*yp2;
1204     iyp = p->yp; iyp2 = i*yp2;
1205     iyp = p->yp; iyp2 = i*yp2;
1206     if ( xleft <= (ixp2) && (ixp2) <= xright &&
1207         ytop <= (iyp2) && (iyp2) <= ybot )
1208     {
1209         circle( (ixp2) - xoffset, ((iyp2) - yoffset), 5 );
1210     }
1211     if ( ( mode == 1 ) && ( i % 2 == 0 ) )
1212     {
1213         if ( xleft <= (liyp2) && (liyp2) <= xright &&
1214             ytop <= (liyp2) && (liyp2) <= ybot )
1215         {
1216             line( (liyp2) - xoffset, (liyp2) - yoffset,
1217                  (ixp2) - xoffset, (iyp2) - yoffset );
1218         }
1219     }
1220 }
1221
1222

```

```

1223     c1x = ((double) i*xp) * pxmag;
1224     c1y = ((double) i*yp) * pythag;
1225     c2x = ((double) lixp) * pxmag;
1226     c2y = ((double) liyp) * pythag;
1227
1228     dx = c2x - c1x;
1229     dy = c2y - c1y;
1230     dist = 10000.0 * sqrt( dy*dy + dx*dx );
1231     /* Factor 10000.0 Converts cms to microns */
1232     if ( show_velocity_on == 1 )
1233     {
1234         velocity = dist / pulse_separation;
1235         printf( string, "%.01f", velocity );
1236     }
1237     else
1238     {
1239         if ( dx != 0.0 )
1240             angle = ( atan2( (double) dy, (double) dx ) / 3.141592654 ) *
1241                 180.0;
1242         else
1243             angle = 90.0;
1244         angle += 270.0;
1245         if ( angle >= 360.0 ) angle -= 360.0;
1246         if ( angle >= 180.0 ) angle -= 180.0;
1247         printf( string, " %.01f", angle );
1248     }
1249     outtextxy( (liyp2) - xoffset, (liyp2) - yoffset, string );
1250 }
1251
1252 p = p->succ;
1253
1254 direction = Arrow_Key();
1255 if ( direction == 5 ) Save_Screen( "zoomdin.raw" );
1256 if ( direction == 6 )
1257 {
1258     if ( tif_display_on == 1 ) tif_display_on = 0;
1259     else tif_display_on = 1;
1260 }
1261 if ( direction == 7 )
1262 {
1263     if ( show_velocity_on == 1 ) show_velocity_on = 0;
1264     else show_velocity_on = 1;
1265 }
1266 cleardevice();
1267 switch( direction ) {
1268     case 1 : if ( yoffset >= yinc ) yoffset -= yinc; break;
1269     case 2 : if ( (yoffset-getmaxy()) < llen ) yoffset += yinc; break;
1270 }

```

PIDV	/ap/main.c	Page 21
1288	case 3 : if (xoffset >= xinc) xoffset -= xinc; break;	
1289	case 4 : if ((xoffset-getmaxx()) < iwid) xoffset += xinc; break;	
1290	default : break;	
1291	}	
1292	}	
1293	} while(direction != 0) ;	
1294	}	
1295		
1296		
1297		
1298	Free_Bit_Pixel_Buffer(bit_buffer, (unsigned long) (yblocksize-overlap));	
1299		
1300	closegraph();	
1301	restorecrtmode();	
1302		
1303	Print_Title(title); Main_Menu(tmain);	
1304		
1305		
1306	else Pop_Error("Tif file is not open");	
1307	}	
1308	else Pop_Error(grapherrormsg(errorcode));	
1309	}	
1310	else Pop_Error("Particle Buffer is Empty");	
1311	}	
1312		
1313		
1314	void Display_PIV_Particles()	
1315	{	
1316	Display_PIV(0);	
1317	}	
1318		
1319	void Display_PIV_Particles_Paired()	
1320	{	
1321	Display_PIV(1);	
1322	}	
1323		
1324	void Display_Zoomed_Out(mode)	
1325	int mode;	
1326	{	
1327	FILE* in;	
1328	int x,y;	
1329	/* request auto detection */	
1330	int gdriver = DETECT, gmode, errorcode;	
1331	int c;	
1332	long i;	
1333	long last_particle;	
1334	PARTICLE_POINTER p;	
1335	char string[80];	
1336	float scalex, scaley;	
1337	int lxp,lyp;	
1338	int lxp, llyp;	
1339	int key;	
1340	int xasp;	
1341	int yasp;	
1342	int xleft,xright;	
1343	int ytop,ybot;	
1344		
1345	last_particle = Last_Particle();	
1346		
1347	if (last_particle != 0) {	
1348	gmode = VGAHI;	
1349		
1350	/* initialize graphics mode */	
1351		
1352	if (mode < 2) {	
1353		

PIDV	/ap/main.c	Page 22
1354	initgraph(&gdriver, &gmode, "");	
1355		
1356	/* read result of initialization */	
1357		
1358	errorcode = graphresult();	
1359		
1360	/* Set_Block_Palette(1) */	
1361		
1362		
1363		
1364	else errorcode = grOk;	
1365		
1366	if (errorcode == grOk)	
1367	{	
1368	if (tif != NULL)	
1369	{	
1370		
1371	getaspectratio(&xasp, &yasp);	
1372		
1373	if ((imagewidth > imagelength) &&	
1374	(xasp >= yasp) &&	
1375	(getmaxx() >= getmaxy()))	
1376	{	
1377	scalex= ((float) xasp) * ((float) getmaxx()) / (((float) imagewidth)*10000.	
1378	0);	
1379	scaley= ((float) yasp) * ((float) getmaxy()) / (((float) imagewidth)*10000.	
1380	0);	
1381	else	
1382	{	
1383	scalex= ((float) xasp) * ((float) getmaxx()) / (((float) imagelength)*10000	
1384	0);	
1385	scaley= ((float) yasp) * ((float) getmaxy()) / (((float) imagelength)*10000	
1386	0);	
1387	setcolor(getmaxcolor());	
1388		
1389	xleft = 0;	
1390	xright = (int) ((scalex * ((float) imagewidth-1)) + 0.5);	
1391	ytop = 0;	
1392	ybot = (int) ((scaley * ((float) imagelength-1)) + 0.5);	
1393		
1394	line(xleft, ytop, xright, ytop);	
1395	line(xright, ytop, xright, ybot);	
1396	line(xright, ybot, xleft, ybot);	
1397	line(xleft, ybot, xleft, ytop);	
1398		
1399	lxp = llyp = -1;	
1400	p = Particles.head->succ;	
1401		
1402	while (p != Particles.head)	
1403	{	
1404	lxp = p->xp;	
1405	lyp = p->yp;	
1406		
1407	p = p->succ;	
1408		
1409	if ((p != Particles.head) && ((mode & 2) == 1)) {	
1410		
1411	lxp = p->xp;	
1412	llyp = p->yp;	
1413		
1414	setcolor(EGA_CYAN);	
1415		

PIDV	/ap/main.c	Page 23
1416	line((int) (((float) (ixp/2)) * scalex + 0.5),	
1417	(int) (((float) (iyp/2)) * scaley + 0.5),	
1418	(int) (((float) (ixp/2)) * scalex + 0.5),	
1419	(int) (((float) (iyp/2)) * scaley + 0.5),	
1420	(int) (((float) (ixp/2)) * scalex + 0.5),	
1421	(int) (((float) (iyp/2)) * scaley + 0.5),	
1422	p = p->succ;	
1423		
1424		
1425	else putpixel((int) (((float) ixp) * scalex + 0.5),	
1426	(int) (((float) iyp) * scaley + 0.5), 63);	
1427	};	
1428		
1429	key = Get_Next_Char();	
1430	if (toupper(key) == 'S') Save_Screen();	
1431		
1432	closegraph();	
1433	restorecrtmode();	
1434		
1435	Print_Title(title); Main_Menu(&Main);	
1436		
1437	else Pop_Error("Tif file is not open");	
1438		
1439	else Pop_Error("grapherrormsg(errorcode));	
1440		
1441	else Pop_Error("Particle Buffer is Empty");	
1442		
1443		
1444		
1445		
1446	void Display_PIV_Particles_Zoomed_Out()	
1447	{	
1448	Display_Zoomed_Out(1);	
1449	}	
1450		
1451	void Write_Particle_Buffer_Text()	
1452	{	
1453	FILE* out;	
1454	long i;	
1455	long last;	
1456	PARTICLE_POINTER p;	
1457	double xmic;	
1458	double ymic;	
1459	char frame[80];	
1460	sprintf(frame, "%s.txt", piv_prefix);	
1461	out = fopen(frame, "w");	
1462		
1463	if (out != NULL)	
1464	{	
1465	last = Last_Particle();	
1466	for(i = 1; i <= last; i++)	
1467	{	
1468	p = Set_Curr_Particle(i);	
1469		
1470	xmic = ((double) p->xp) * p->mag;	
1471	ymic = ((double) p->yp) * p->mag;	
1472		
1473	fprintf(out, "Particle No. %ld\n", i);	
1474	fprintf(out, "Centre (Tif Coordinates): x=%ld y=%ld\n", (p->	
1475	xp/2), (p->yp/2));	
1476	fprintf(out, "Centre (Scaled to Microns): x mu m=%ld, y mu m=	
1477	%ld\n", xmic,ymic);	
1478	fprintf(out, "Area of Particle: %ld pixels\n",p->mass);	
1479		

PIDV	/ap/main.c	Page 24
1480	fprintf(out, "\n");	
1481	}	
1482		
1483	fclose(out);	
1484	}	
1485		
1486		
1487	void Write_PIV_Data()	
1488	{	
1489	char frame[80];	
1490	char string[80];	
1491	int c;	
1492	PARTICLE_POINTER p;	
1493	long i;	
1494		
1495	if (Particles.head->succ != Particles.head) {	
1496	sprintf(string,	
1497	"Will overwrite %s.par, ESC to cancel, Enter to continue",	
1498	piv_prefix);	
1499		
1500	Pop_Print(string, 0);	
1501		
1502	while ((c = Get_Next_Char()) != ENTER) { (c != ESCAPE));	
1503	Clear_Print();	
1504		
1505	if (c == ENTER) {	
1506		
1507	sprintf(frame, "%s.par", piv_prefix);	
1508		
1509	Create_Particle_Data_File(frame);	
1510		
1511	Write_Particle_Buffer(frame);	
1512		
1513	}	
1514		
1515		
1516		
1517		
1518		
1519	void Read_PIV_Data(particle_file_name)	
1520	char* particle_file_name;	
1521	{	
1522	if (Last_Particle() == 0) {	
1523		
1524	Read_Particle_Buffer(particle_file_name);	
1525		
1526	} else Pop_Error("Current Particle Buffer Must Be Empty!");	
1527		
1528		
1529	void Read_PIV_Data_Par()	
1530	{	
1531	char file_name[80];	
1532		
1533	if (strcmp(piv_prefix, "NOT SET") != 0) {	
1534	sprintf(file_name, "%s.par", piv_prefix);	
1535		
1536	Read_PIV_Data(file_name);	
1537		
1538		
1539	} else Pop_Error("Must Open a TIFF File First!");	
1540		
1541		
1542	void Write_Feature_Plot_File(Buffer, frame, matlab_mode)	
1543	Buffer* Buffer;	
1544	char* frame;	
1545	int matlab_mode;	

PIDV	./ap/main.c	Page 26
1612	fclose(out);	
1613	}	
1614	else Pop_Error("Couldn't Open Feature Plot File");	
1615	}	
1616	}	
1617	}	
1618	}	
1619	void Write_Matlab_Plot_File(fname, matlab_mode)	
1620	{	
1621	char* fname;	
1622	int matlab_mode;	
1623	{	
1624	FILE* out;	
1625	long i;	
1626	long last;	
1627	PARTICLE_POINTER p1;	
1628	PARTICLE_POINTER p2;	
1629	double x1,y1,x2,y2;	
1630	double midx, midy;	
1631	double velocity,angle;	
1632	double dx,dy;	
1633	double dist;	
1634	FMatrix mat;	
1635	char* pname;	
1636	double d[4];	
1637	if (matlab_mode == 1)	
1638	{	
1639	mat.type = 0L; /* Zero for PC */	
1640	mat.ncols = 4L;	
1641	mat.mrows = 4L;	
1642	mat.ncols = (long) (Last_Particle() + 1);	
1643	mat.mrows = 0L;	
1644	mat.ncols = 0L;	
1645	mat.mrows = 0L;	
1646	mat.ncols = 0L;	
1647	mat.mrows = 0L;	
1648	if ((out = fopen(fname, "wb")) != NULL) {	
1649	{	
1650	fwrite(&mat, sizeof(FMatrix), 1, out);	
1651	fwrite(pname, sizeof(char), (int)mat.mrows, out);	
1652	i = 1;	
1653	last = Last_Particle();	
1654	do {	
1655	{	
1656	p1 = Set_Curr_Particle(i);	
1657	p2 = Set_Curr_Particle(i+1);	
1658	x1 = ((double) p1->x) * pxmag * 10000.0;	
1659	y1 = ((double) p1->y) * pythag * 10000.0;	
1660	x2 = ((double) p2->x) * pxmag * 10000.0;	
1661	y2 = ((double) p2->y) * pythag * 10000.0;	
1662	dx = (x2 - x1);	
1663	dy = (y2 - y1);	
1664	dist = sqrt(dx*dx + dy*dy);	
1665	velocity = dist / pulse_separation;	
1666	if (dx != 0.0)	
1667	angle = (atan2((double) dy, (double) dx) / 3.141592654) * 180.0	
1668	else angle = 90.0;	
1669	}	
1670	i++;	
1671	}	
1672	}	
1673	}	
1674	}	
1675	}	
1676	}	

PIDV	./ap/main.c	Page 25
1546	{	
1547	FILE* out;	
1548	int PIXEL* p;	
1549	float x,y;	
1550	FMatrix mat;	
1551	char* pname;	
1552	double d[4];	
1553	if (matlab_mode == 1)	
1554	{	
1555	mat.type = 0L; /* Zero for PC */	
1556	mat.ncols = (long) Last(Buffer);	
1557	mat.mrows = 2L;	
1558	mat.ncols = 0L;	
1559	mat.mrows = 0L;	
1560	mat.ncols = 0L;	
1561	mat.mrows = 0L;	
1562	mat.ncols = 0L;	
1563	mat.mrows = 0L;	
1564	if ((out = fopen(fname, "wb")) != NULL) {	
1565	{	
1566	fwrite(&mat, sizeof(FMatrix), 1, out);	
1567	fwrite(pname, sizeof(char), (int)mat.mrows, out);	
1568	p = Buffer->head->succ;	
1569	while(p != Buffer->head)	
1570	{	
1571	d[0] = (double) ((float) p->x*2) * pxmag * 10000.0;	
1572	d[1] = (double) ((float) p->y*2) * pythag * 10000.0;	
1573	d[2] = (double) ((float) p->x*2) * pxmag * 10000.0;	
1574	d[3] = (double) ((float) p->y*2) * pythag * 10000.0;	
1575	fwrite((void*) d, sizeof(double), (int) 1, out);	
1576	p = p->succ;	
1577	}	
1578	}	
1579	}	
1580	}	
1581	}	
1582	}	
1583	}	
1584	}	
1585	}	
1586	}	
1587	}	
1588	}	
1589	}	
1590	}	
1591	}	
1592	}	
1593	}	
1594	}	
1595	}	
1596	}	
1597	}	
1598	}	
1599	}	
1600	}	
1601	}	
1602	}	
1603	}	
1604	}	
1605	}	
1606	}	
1607	}	
1608	}	
1609	}	
1610	}	
1611	}	

PIDV	./ap/main.c	Page 28
1742	midx = ((x2 - x1) / 2.0) + x1;	
1743	midy = ((y2 - y1) / 2.0) + y1;	
1744		
1745	fprintf(out, "%f %f %f\n", (float)midx, (float)midy, (float)velocity, (float)angle);	
1746		
1747	i += 2;	
1748	} while (i <= last);	
1749	}	
1750	fprintf(out, "%f %f %f\n",	
1751	(float)pulse_separation,	
1752	(float)vec_scale,	
1753	(float)0.0,	
1754	(float)0.0);	
1755	fclose(out);	
1756	}	
1757	else Pop_Error("Couldn't Open Matlab Plot File");	
1758	}	
1759	}	
1760	}	
1761	}	
1762	}	
1763	}	
1764	void Write_Plot_File()	
1765	{	
1766	char frame[256];	
1767	sprintf(frame, "%s.dat", piv_prefix);	
1768	Write_Matlab_Plot_File(frame, 0);	
1769	}	
1770	}	
1771	void Min_Vel()	
1772	{	
1773	char string[256];	
1774	char* in;	
1775	printf(string, "V = %lf", minimum_velocity);	
1776	if ((in = Pop_Input(string)) != NULL) {	
1777	minimum_velocity = atof(in);	
1778	}	
1779	}	
1780	void Max_Vel()	
1781	{	
1782	char* in;	
1783	printf(string, "V = %lf", maximum_velocity);	
1784	if ((in = Pop_Input(string)) != NULL) {	
1785	maximum_velocity = atof(in);	
1786	}	
1787	}	
1788	void Min_Dir()	
1789	{	
1790	char string[256];	
1791	char* in;	
1792	double t;	
1793	printf(string, "A = %lf", minimum_direction);	
1794	if ((in = Pop_Input(string)) != NULL) {	
1795	minimum_direction = atof(in);	
1796	}	
1797	}	
1798	}	
1799	}	
1800	}	
1801	}	
1802	}	
1803	}	
1804	}	
1805	}	
1806	}	

PIDV	./ap/main.c	Page 27
1677	angle += 270.0;	
1678		
1679	if (angle >= 360.0) angle -= 360.0;	
1680		
1681	if (angle >= 180.0) angle -= 180.0;	
1682		
1683	midx = ((x2 - x1) / 2.0) + x1;	
1684	midy = ((y2 - y1) / 2.0) + y1;	
1685		
1686	d[0] = midx;	
1687	d[1] = midy;	
1688	d[2] = velocity;	
1689	d[3] = angle;	
1690		
1691	fwrite(d, sizeof(double), 4, out);	
1692		
1693	i += 2;	
1694	} while (i <= last);	
1695	}	
1696	}	
1697	}	
1698	}	
1699	}	
1700	}	
1701	}	
1702	}	
1703	}	
1704	}	
1705	}	
1706	}	
1707	}	
1708	}	
1709	}	
1710	}	
1711	}	
1712	}	
1713	}	
1714	}	
1715	}	
1716	}	
1717	}	
1718	}	
1719	}	
1720	}	
1721	}	
1722	}	
1723	}	
1724	}	
1725	}	
1726	}	
1727	}	
1728	}	
1729	}	
1730	}	
1731	}	
1732	}	
1733	}	
1734	}	
1735	}	
1736	}	
1737	}	
1738	}	
1739	}	
1740	}	
1741	}	

PIDV	.lap/main.c	Page 29
1807	t = atof(in);	
1808	if ((t < 0.0) (t > 180.0))	
1809	Pop_Error("Out of Range, (0 -> 180)");	
1810	else minimum_direction = t;	
1811		
1812		
1813		
1814		
1815		
1816		
1817	void Max_Dir()	
1818	{	
1819	char* in;	
1820	char string[256];	
1821	double t;	
1822	printf(string, "A = %lf", maximum_direction);	
1823	if ((in = Pop_Input(string)) != NULL) {	
1824	t = atof(in);	
1825	if ((t < 0.0) (t > 180.0))	
1826	Pop_Error("Out of Range, (0 -> 180)");	
1827	else maximum_direction = t;	
1828		
1829		
1830		
1831		
1832		
1833		
1834		
1835	void Set_Min_Psize()	
1836	{	
1837	char* num;	
1838	char string[256];	
1839	int n;	
1840	printf(string, "Min = %ld", min_particle_size);	
1841	if ((num = Pop_Input(string)) != NULL)	
1842	{	
1843	previous_min_particle_size = min_particle_size;	
1844	min_particle_size = atoi(num);	
1845		
1846		
1847		
1848		
1849		
1850	void Set_Min_Feature_Size()	
1851	{	
1852	char* num;	
1853	char string[256];	
1854	int n;	
1855	printf(string, "Min = %ld", min_feature_size);	
1856	if ((num = Pop_Input(string)) != NULL)	
1857	{	
1858	previous_min_feature_size = min_feature_size;	
1859	min_feature_size = atoi(num);	
1860		
1861		
1862		
1863		
1864		
1865	void Set_Max_Psize()	
1866	{	
1867	char* num;	
1868	char string[256];	
1869	int n;	
1870	printf(string, "Max = %ld", max_particle_size);	
1871		
1872		

PIDV	.lap/main.c	Page 30
1873	if ((num = Pop_Input(string)) != NULL)	
1874	{	
1875	previous_max_particle_size = max_particle_size;	
1876	max_particle_size = atoi(num);	
1877		
1878		
1879		
1880	void Set_Box_Size()	
1881	{	
1882	char* num;	
1883	char string[256];	
1884	int n;	
1885	printf(string, "%.2lf, Microns", box_size_microns);	
1886	if ((num = Pop_Input(string)) != NULL)	
1887	{	
1888	box_size_microns = atof(num);	
1889	box_size_x = (int) ((box_size_microns / (pymag*2.0*10000.0)) + 0.5);	
1890	box_size_y = (int) ((box_size_microns / (pymag*2.0*10000.0)) + 0.5);	
1891		
1892		
1893		
1894		
1895		
1896		
1897	void Set_Scale_For_Lcm_In_Scanned_Image()	
1898	{	
1899	char* num;	
1900	char string[256];	
1901	int n;	
1902	printf(string, "1 cm = %.4lf", one_cm_scale);	
1903	if ((num = Pop_Input(string)) != NULL)	
1904	{	
1905	one_cm_scale = (double) atof(num);	
1906	switch (td->td_resolutionunit) {	
1907	case RESUNIT_NONE:	
1908	Pop_Error("Don't Know The Resolution Unit Of The Scanned Image!");	
1909	break;	
1910	case RESUNIT_INCH:	
1911	pymag = (one_cm_scale * INCHES_TO_CMS / (td->td_resolution * 2.0))	
1912	};	
1913		
1914		
1915		
1916		
1917		
1918		
1919	break;	
1920	case RESUNIT_CENTIMETER:	
1921	pymag = (one_cm_scale / (td->td_resolution * 2.0));	
1922	pymag = (one_cm_scale / (td->td_resolution * 2.0));	
1923	break;	
1924	default : Pop_Error("Don't Know The Resolution Unit Of The Scanned Imag	
1925	e!");	
1926	break;	
1927		
1928		
1929	void Set_Pulse_Separation()	
1930	{	
1931	char* num;	
1932	char string[80];	
1933	int n;	
1934		
1935		

PIDV	/ap/main.c	Page 31
1336	printf(string, "Sep = %.4lf", pulse_separation);	
1337	if ((num = Pop_Input(string)) != NULL)	
1338	{	
1339	pulse_separation = atof(num);	
1340	};	
1341		
1342		
1343	void Set_Vec_Scale()	
1344	{	
1345	char* num;	
1346	char string[80];	
1347	int n;	
1348	};	
1349		
1350	printf(string, "Scale = %.2lf", vec_scale);	
1351	if ((num = Pop_Input(string)) != NULL)	
1352	{	
1353	vec_scale = atof(num);	
1354	};	
1355		
1356		
1357	void Set_Intensity_Threshold()	
1358	{	
1359	char* num;	
1360	char string[256];	
1361	int n;	
1362	};	
1363	printf(string, "Threshold = %.1lf%%", (intensity_threshold*100.0));	
1364	if ((num = Pop_Input(string)) != NULL)	
1365	{	
1366	intensity_threshold = atof(num) / 100.0;	
1367	eight_bit_intensity_threshold = (int) (255.0 * intensity_threshold + 0.5)	
1368	};	
1369		
1370		
1371		
1372		
1373	void Save_PIV_Config_File(fname)	
1374	char* fname;	
1375	{	
1376	FILE* cfg;	
1377	};	
1378	cfg = fopen(fname, "w");	
1379	if (cfg != NULL)	
1380	{	
1381	fprintf(cfg, "%ld,%ld,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%ld,%ld,%ld\n",	
1382	min_particle_size,max_particle_size,	
1383	one_cm_scale,pxmag,pymag,pulse_separation,	
1384	minimum_velocity,maximum_velocity,	
1385	minimum_direction,maximum_direction,	
1386	intensity_threshold,eight_bit_intensity_threshold,	
1387	vec_scale,box_size_microns,box_size_x,box_size_y,	
1388	min_feature_size,write_coordinates	
1389);	
1390	};	
1391	fprintf(cfg, "%s\n", piv_prefix);	
1392	fclose(cfg);	
1393		
1394		
1395		
1396	else Pop_Error("Couldn't open PIV configuration file",1);	
1397		
1398		
1399		

PIDV	/ap/main.c	Page 32
2000	int Load_PIV_Config_File(fname)	
2001	char* fname;	
2002	{	
2003	FILE* cfg;	
2004	int i;	
2005	int success = 1;	
2006	char cfg_data[256];	
2007	};	
2008	cfg = fopen(fname, "r");	
2009	if (cfg != NULL)	
2010	{	
2011	fgets(cfg_data, 256, cfg);	
2012	};	
2013		
2014	fscanf(cfg_data, "%ld,%ld,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%ld,%ld,%ld\n",	
2015	min_particle_size,max_particle_size,	
2016	one_cm_scale,pxmag,pymag,pulse_separation,	
2017	minimum_velocity,maximum_velocity,	
2018	minimum_direction,maximum_direction,	
2019	intensity_threshold,eight_bit_intensity_threshold,	
2020	vec_scale,box_size_microns,box_size_x,box_size_y,	
2021	min_feature_size,write_coordinates	
2022);	
2023	};	
2024	};	
2025	};	
2026		
2027		
2028		
2029	previous_min_particle_size = min_particle_size;	
2030	previous_max_particle_size = max_particle_size;	
2031	previous_min_feature_size = min_feature_size;	
2032	fclose(cfg);	
2033		
2034	else success = 0;	
2035	};	
2036	return(success);	
2037		
2038		
2039	void Del_Particle_From_Buffer()	
2040	{	
2041	char* num;	
2042	char string[256];	
2043	long n;	
2044	};	
2045	printf(string, "NP = %ld, Enter particle No.", Last_Particle());	
2046	if ((num = Pop_Input(string)) != NULL)	
2047	{	
2048	n = atoi(num);	
2049	if (n <= Last_Particle()) (void) Del_Particle(n);	
2050	};	
2051		
2052		
2053	void Read_PIV_Parameter_Data_File()	
2054	{	
2055	char fname[256];	
2056	};	
2057	printf(fname, "%s.cfg", piv_prefix);	
2058		
2059	if (Load_PIV_Config_File(fname) != 1)	
2060	{	
2061	printf(fname, "Cannot open PIV parameter file %s.cfg", piv_prefix);	
2062	Pop_Print(fname, 1);	
2063	};	
2064		

```

2065 }
2066
2067 void Write_PIV_Parameter_Data_File()
2068 {
2069     char fname[256];
2070     sprintf( fname, "%s.cfg", piv_prefix );
2071     Save_PIV_Config_File( fname );
2072 }
2073
2074 void Batch_Process()
2075 {
2076     char* inp;
2077     FILE* batch;
2078     char tif_file[80];
2079     unsigned long core_memory;
2080     char string[80];
2081     if ( ( in = Pop_Input( "Enter batch file name" ) ) != NULL )
2082     {
2083         batch = fopen( in, "r" );
2084         if ( batch != NULL )
2085         {
2086             while( fgets( tif_file, 80, batch ) != NULL )
2087             {
2088                 if ( kbhit() ) break;
2089                 tif_file[ strlen( tif_file ) - 1 ] = '\0';
2090                 if ( tif != NULL ) TIFFClose( tif );
2091                 tif = NULL;
2092                 core_memory = coreleft();
2093                 sprintf( string, "Core Memory Left %lu Bytes", core_memory );
2094                 Pop_Print( string, 0 );
2095                 Open_PIV_Tif_File( tif_file );
2096                 Clear_Print();
2097                 Process_Piv();
2098                 Clear_Print();
2099                 fclose( batch );
2100             }
2101         }
2102     }
2103     void Quit()
2104     {
2105         int c;
2106         char frame[80];
2107         if ( tif != NULL ) TIFFClose( tif );
2108         fcloseall();
2109         if ( strcmp( piv_prefix, "NOT SET" ) != 0 ) {

```

```

2131     sprintf( fname, "%s.cfg", piv_prefix );
2132     Save_PIV_Config_File( fname );
2133 }
2134
2135 Set_Test_Page( 0 );
2136 CIs();
2137 exit( 0 );
2138 }
2139
2140 void Core_Left()
2141 {
2142     unsigned long core_memory;
2143     char string[80];
2144     core_memory = coreleft();
2145     sprintf( string, "Core Memory Left %lu Bytes", core_memory );
2146     Pop_Print( string, 1 );
2147 }
2148
2149 void Open_PIV_Tif_File( fname )
2150 {
2151     int i;
2152     char string[80];
2153     strcpy( tif_file_name, fname );
2154     if ( tif != NULL ) TIFFClose( tif );
2155     tif = TIFFOpen( tif_file_name, "r" );
2156     Pop_Print( "" );
2157     Clear_Print();
2158     if ( tif == NULL ) Pop_Error( "Unable to open TIF file" );
2159     else {
2160         int i=0;
2161         TIFFGetField( tif, TIFFTAG_IMAGELENGTH, &imagelength );
2162         while( ( tif_file_name[i] != '.' ) && ( tif_file_name[i] != '\0' ) )
2163             i++;
2164         strcpy( piv_prefix, tif_file_name );
2165         piv_prefix[i] = '\0';
2166         Set_Title( piv_prefix ); td = &tif->tif_dir;
2167         imagewidth = td->td_imagewidth;
2168         if ( raster != NULL ) Free( raster );
2169         raster_size = TIFFScanlinesize( tif );
2170         raster = Malloc( (unsigned long) raster_size );
2171         if ( raster == NULL )
2172             Pop_Error( "Couldn't Allocate Memory For Raster Buffer" );
2173         sprintf( string, "%s.cfg", piv_prefix );
2174     }
2175 }

```



```

2197 if ( Load_PIV_Config_File( string ) == 0 )
2198     scale_factors_not_set = 1;
2199 else scale_factors_not_set = 0;
2200 Delete_All_Particles();
2201 Pop_Print("Loading Particle Data..", 0 );
2202 sprintf( string, "%s.prs", piv_prefix );
2203 Read_PIV_Data( string );
2204 Clear_Print();
2205 }
2206 void Open_Tif_File()
2207 {
2208     char* in;
2209     int i,l;
2210     char string[80];
2211     if ( ( in = Pop_Input("Enter tif file prefix") ) != NULL )
2212     {
2213         sprintf( string, "%s.tif", in );
2214         Open_PIV_Tif_File( string );
2215     }
2216     void Read_All_PIV_Files()
2217     {
2218         char fname[80];
2219         sprintf( fname, "%s.tif", piv_prefix );
2220         Open_PIV_Tif_File( fname );
2221         sprintf( fname, "%s.cfg", piv_prefix );
2222         if ( Load_PIV_Config_File( fname ) == 1 ) {
2223             sprintf( fname, "%s.prs", piv_prefix );
2224             if ( Last_Particle() == 0 ) {
2225                 Read_Particle_Buffer(fname);
2226             } else Pop_Print("Current Particle Buffer Must Be Empty!", 1 );
2227         } else {
2228             sprintf( fname, "Cannot open PIV parameter file %s.cfg", piv_prefix );
2229             Pop_Print(fname, 1 );
2230         }
2231     }
2232     int matherr( struct exception *s )
2233     {
2234         Pop_Error("Floating Point Exception, Press ESC to ignore and continue", 1);
2235         return( 1 );
2236     }
2237     void Start_Up()
2238     {
2239         char fname[80];
2240         tif = NULL;
2241     }
2242

```

```

2263 Initialise_Pop_Up_Menus();
2264 raster = NULL;
2265 raster_size = 0;
2266 bit_buffer_width = 0;
2267 Bfill = (BUFF*) Malloc( (unsigned long) ( sizeof( BUFF ) ) );
2268 if ( Bfill == NULL ) Pop_Error( "Ran out of memory allocating flood fill buffer", 1 );
2269 if ( Bfill == NULL ) Pop_Error( "Ran out of memory allocating edge fill buffer", 1 );
2270 Bedge = (BUFF*) Malloc( (unsigned long) ( sizeof( BUFF ) ) );
2271 if ( Bedge == NULL ) Pop_Error( "Ran out of memory allocating edge fill buffer", 1 );
2272 Init_Pixel_Buffer( Bfill );
2273 Init_Pixel_Buffer( Bedge );
2274 Init_Particle_Buffer();
2275 sprintf( piv_prefix, "NOT SET" );
2276 Set_Title( piv_prefix );
2277 particle_pixels = 0;
2278 min_particle_size = 1;
2279 min_feature_size = 1000;
2280 max_particle_size = 100;
2281 previous_min_particle_size = -1;
2282 previous_max_particle_size = -1;
2283 previous_min_feature_size = -1;
2284 one_cm_scale = 1.0;
2285 pxmag = 0.0;
2286 minimum_velocity = 0.0; /* Metres per second */
2287 maximum_velocity = 660.0; /* Metres per second */
2288 minimum_direction = 0.0;
2289 maximum_direction = 180.0;
2290 intensity_threshold = 0.5; /* 50 percent */
2291 eight_bit_intensity_threshold = (int) (255.0 * intensity_threshold * 0.5);
2292 pulse_separation = 1.0; /* Microseconds */
2293 scale_factor_not_set = 1;
2294 vec_scale = 1.0;
2295 dir_grid = NULL;
2296 box_size_x = 0;
2297 box_size_y = 0;
2298 box_size_micros = 1000.0;
2299 nbox_x = 1;
2300 nbox_y = 1;
2301 write_coordinates = 1;
2302 Define_Menus();
2303 main( argc, argv )
2304 int argc;
2305 char* argv[];
2306 {
2307     Start_Up();
2308 }

```

PIDV	/ap/main.c	Page 37
2327		
2328	if (argc == 2) Open_PIV_Tif_File(argv[1]);	
2329	for(; ;) { Print_Title(title); Main_Menu(\$Main); }	
2330		
2331	}	

PIDV	/ap/memory.h	Page 1
1	/*	
2	* Copyright (c) 1991 by Tom Judge.	
3	* All rights reserved.	
4	*	
5	*/	
6	extern void* Malloc();	
7	extern void Free();	
8	extern void Calloc();	

```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  #include <alloc.h>
6
7  void* Malloc( size_t
8  unsigned long size;
9
10 {
11     return( (void*) malloc( (size_t) size ) );
12 }
13
14 void Free( block )
15 void* block;
16 {
17     free( block );
18 }
```

```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  extern void Write_Pixel_Buffer();
6  extern void Write_Particle_Buffer();
7  extern void Read_Pixel_Buffer();
8  extern void Particle_Size();
9  extern void Process_Piv();
10 extern void Delete_Rogue_Particles();
11 extern void Set_Title();
12 extern void Print_File_Dir();
13 extern void Display_File();
14 extern void Display_Min_Dir();
15 extern void Display_Piv();
16 extern void Display_Piv_Particles();
17 extern void Display_Zoomed_Out();
18 extern void Display_Zoomed_Out();
19 extern void Display_Piv_Particles_Zoomed_Out();
20 extern void Display_Piv_Particles_Zoomed_Out();
21 extern void Write_Particle_Buffer_Text();
22 extern void Read_Whole_Particle_Buffer();
23 extern void Read_Read_Piv_Data();
24 extern void Read_Read_Piv_Data_Par();
25 extern void Write_Particle_Pair_Velocities();
26 extern void Write_Feature_Plot_File();
27 extern void Write_Matlab_Plot_File();
28 extern void Min_Vel();
29 extern void Max_Vel();
30 extern void Min_Dir();
31 extern void Max_Dir();
32 extern void Min_Pair();
33 extern void Max_Pair();
34 extern void Set_Min_Feature_Size();
35 extern void Set_Max_Pair();
36 extern void Set_Box_Size();
37 extern void Set_Scale_For_1cm_In_Scanned_Image();
38 extern void Set_Pulse_Separation();
39 extern void Set_Vac_Scale();
40 extern void Set_Intensity_Threshold();
41 extern void Save_Piv_Config_File();
42 extern void Del_Particle_From_Buffer();
43 extern void Read_Piv_Parameter_Data_File();
44 extern void Write_Piv_Parameter_Data_File();
45 extern void Batch_Process();
46 extern void Quit();
47 extern void Core_Let();
48 extern void Open_Piv_File();
49 extern void Open_All_Files();
50 extern void Read_All_Files();
51 extern void Start_Up();
52
53
54 extern void Define_Menus();
55
56 extern MENU_Main;
57 extern MENU_File;
58 extern MENU_Piv;
59 extern MENU_Piv_Scale;
```

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  #include <stdio.h>
6  #include <math.h>
7  #include <graphics.h>
8  #include <alloc.h>
9  #include <dos.h>
10 #include <values.h>
11 #include <time.h>
12 #include <assert.h>
13 #include <stdio.h>
14 #include "p1st.h"
15 #include "memory.h"
16 #include "dgrid.h"
17 #include "main.h"
18 #include "pivfiles.h"
19
20 int Create_Particle_Data_File( fname )
21 char* fname;
22 {
23     FILE* out;
24     int result = 1;
25
26     if ( fname != NULL )
27     {
28         out = fopen( fname , "wb" );
29         if ( out == NULL )
30             result = 0;
31         fclose( out );
32     }
33     else
34     {
35         Pop_Error( "Unable to Create Particle Data File!" );
36         result = 0;
37     }
38
39     return( result );
40 }
41
42 int Create_Feature_Data_File( fname )
43 char* fname;
44 {
45     FILE* out;
46     int result = 1;
47
48     if ( fname != NULL )
49     {
50         out = fopen( fname , "wb" );
51         if ( out == NULL )
52             result = 0;
53         fclose( out );
54     }
55     else
56     {
57         Pop_Error( "Unable to Create Feature Data File!" );
58         result = 0;
59     }
60
61     return( result );
62 }
63
64 void Write_Pixel_Buffer( Buffer, fname )
65 char* Buffer;
66 char* fname;
67 {
68     XY_PIXEL* p;
69
70     if ( ! Create_Feature_Data_File( fname ) )
71         return;
72
73     p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
74
75     if ( ! p )
76         return;
77
78     while ( ! feof( input ) )
79     {
80         p->x = getw( input );
81         p->y = getw( input );
82         Push_Pixel( p, Buffer );
83         Free_Pixel( p );
84     }
85
86     fclose( input );
87
88     void Write_Pixel_Buffer( Buffer, fname )
89     char* Buffer;
90     {
91         FILE* input;
92         XY_PIXEL* p;
93         int ret;
94
95         if ( ! Create_Feature_Data_File( fname ) )
96             return;
97
98         input = fopen( fname , "rb" );
99         if ( ! input )
100             return;
101
102         do
103         {
104             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
105             if ( ! p )
106                 return;
107
108             ret = feof( input );
109             if ( ret == 0 )
110             {
111                 p->x = getw( input );
112                 p->y = getw( input );
113                 Push_Pixel( p, Buffer );
114                 Free_Pixel( p );
115             }
116             else
117                 while ( ret == 0 )
118                     continue;
119         } while ( ! feof( input ) );
120
121         fclose( input );
122     }
123
124     void Write_Pixel_Buffer( Buffer, fname )
125     char* Buffer;
126     {
127         XY_PIXEL* p;
128         FILE* out;
129
130         out = fopen( fname, "wb" );
131         if ( ! out )
132             return;
133
134         while ( ! feof( input ) )
135         {
136             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
137             if ( ! p )
138                 return;
139
140             ret = feof( input );
141             if ( ret == 0 )
142             {
143                 p->x = getw( input );
144                 p->y = getw( input );
145                 Push_Pixel( p, Buffer );
146                 Free_Pixel( p );
147             }
148             else
149                 while ( ret == 0 )
150                     continue;
151         } while ( ! feof( input ) );
152
153         fclose( input );
154     }
155
156     void Write_Pixel_Buffer( Buffer, fname )
157     char* Buffer;
158     {
159         XY_PIXEL* p;
160         FILE* out;
161
162         out = fopen( fname, "wb" );
163         if ( ! out )
164             return;
165
166         while ( ! feof( input ) )
167         {
168             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
169             if ( ! p )
170                 return;
171
172             ret = feof( input );
173             if ( ret == 0 )
174             {
175                 p->x = getw( input );
176                 p->y = getw( input );
177                 Push_Pixel( p, Buffer );
178                 Free_Pixel( p );
179             }
180             else
181                 while ( ret == 0 )
182                     continue;
183         } while ( ! feof( input ) );
184
185         fclose( input );
186     }
187
188     void Write_Pixel_Buffer( Buffer, fname )
189     char* Buffer;
190     {
191         XY_PIXEL* p;
192         FILE* out;
193
194         out = fopen( fname, "wb" );
195         if ( ! out )
196             return;
197
198         while ( ! feof( input ) )
199         {
200             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
201             if ( ! p )
202                 return;
203
204             ret = feof( input );
205             if ( ret == 0 )
206             {
207                 p->x = getw( input );
208                 p->y = getw( input );
209                 Push_Pixel( p, Buffer );
210                 Free_Pixel( p );
211             }
212             else
213                 while ( ret == 0 )
214                     continue;
215         } while ( ! feof( input ) );
216
217         fclose( input );
218     }
219
220     void Write_Pixel_Buffer( Buffer, fname )
221     char* Buffer;
222     {
223         XY_PIXEL* p;
224         FILE* out;
225
226         out = fopen( fname, "wb" );
227         if ( ! out )
228             return;
229
230         while ( ! feof( input ) )
231         {
232             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
233             if ( ! p )
234                 return;
235
236             ret = feof( input );
237             if ( ret == 0 )
238             {
239                 p->x = getw( input );
240                 p->y = getw( input );
241                 Push_Pixel( p, Buffer );
242                 Free_Pixel( p );
243             }
244             else
245                 while ( ret == 0 )
246                     continue;
247         } while ( ! feof( input ) );
248
249         fclose( input );
250     }
251
252     void Write_Pixel_Buffer( Buffer, fname )
253     char* Buffer;
254     {
255         XY_PIXEL* p;
256         FILE* out;
257
258         out = fopen( fname, "wb" );
259         if ( ! out )
260             return;
261
262         while ( ! feof( input ) )
263         {
264             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
265             if ( ! p )
266                 return;
267
268             ret = feof( input );
269             if ( ret == 0 )
270             {
271                 p->x = getw( input );
272                 p->y = getw( input );
273                 Push_Pixel( p, Buffer );
274                 Free_Pixel( p );
275             }
276             else
277                 while ( ret == 0 )
278                     continue;
279         } while ( ! feof( input ) );
280
281         fclose( input );
282     }
283
284     void Write_Pixel_Buffer( Buffer, fname )
285     char* Buffer;
286     {
287         XY_PIXEL* p;
288         FILE* out;
289
290         out = fopen( fname, "wb" );
291         if ( ! out )
292             return;
293
294         while ( ! feof( input ) )
295         {
296             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
297             if ( ! p )
298                 return;
299
300             ret = feof( input );
301             if ( ret == 0 )
302             {
303                 p->x = getw( input );
304                 p->y = getw( input );
305                 Push_Pixel( p, Buffer );
306                 Free_Pixel( p );
307             }
308             else
309                 while ( ret == 0 )
310                     continue;
311         } while ( ! feof( input ) );
312
313         fclose( input );
314     }
315
316     void Write_Pixel_Buffer( Buffer, fname )
317     char* Buffer;
318     {
319         XY_PIXEL* p;
320         FILE* out;
321
322         out = fopen( fname, "wb" );
323         if ( ! out )
324             return;
325
326         while ( ! feof( input ) )
327         {
328             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
329             if ( ! p )
330                 return;
331
332             ret = feof( input );
333             if ( ret == 0 )
334             {
335                 p->x = getw( input );
336                 p->y = getw( input );
337                 Push_Pixel( p, Buffer );
338                 Free_Pixel( p );
339             }
340             else
341                 while ( ret == 0 )
342                     continue;
343         } while ( ! feof( input ) );
344
345         fclose( input );
346     }
347
348     void Write_Pixel_Buffer( Buffer, fname )
349     char* Buffer;
350     {
351         XY_PIXEL* p;
352         FILE* out;
353
354         out = fopen( fname, "wb" );
355         if ( ! out )
356             return;
357
358         while ( ! feof( input ) )
359         {
360             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
361             if ( ! p )
362                 return;
363
364             ret = feof( input );
365             if ( ret == 0 )
366             {
367                 p->x = getw( input );
368                 p->y = getw( input );
369                 Push_Pixel( p, Buffer );
370                 Free_Pixel( p );
371             }
372             else
373                 while ( ret == 0 )
374                     continue;
375         } while ( ! feof( input ) );
376
377         fclose( input );
378     }
379
380     void Write_Pixel_Buffer( Buffer, fname )
381     char* Buffer;
382     {
383         XY_PIXEL* p;
384         FILE* out;
385
386         out = fopen( fname, "wb" );
387         if ( ! out )
388             return;
389
390         while ( ! feof( input ) )
391         {
392             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
393             if ( ! p )
394                 return;
395
396             ret = feof( input );
397             if ( ret == 0 )
398             {
399                 p->x = getw( input );
400                 p->y = getw( input );
401                 Push_Pixel( p, Buffer );
402                 Free_Pixel( p );
403             }
404             else
405                 while ( ret == 0 )
406                     continue;
407         } while ( ! feof( input ) );
408
409         fclose( input );
410     }
411
412     void Write_Pixel_Buffer( Buffer, fname )
413     char* Buffer;
414     {
415         XY_PIXEL* p;
416         FILE* out;
417
418         out = fopen( fname, "wb" );
419         if ( ! out )
420             return;
421
422         while ( ! feof( input ) )
423         {
424             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
425             if ( ! p )
426                 return;
427
428             ret = feof( input );
429             if ( ret == 0 )
430             {
431                 p->x = getw( input );
432                 p->y = getw( input );
433                 Push_Pixel( p, Buffer );
434                 Free_Pixel( p );
435             }
436             else
437                 while ( ret == 0 )
438                     continue;
439         } while ( ! feof( input ) );
440
441         fclose( input );
442     }
443
444     void Write_Pixel_Buffer( Buffer, fname )
445     char* Buffer;
446     {
447         XY_PIXEL* p;
448         FILE* out;
449
450         out = fopen( fname, "wb" );
451         if ( ! out )
452             return;
453
454         while ( ! feof( input ) )
455         {
456             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
457             if ( ! p )
458                 return;
459
460             ret = feof( input );
461             if ( ret == 0 )
462             {
463                 p->x = getw( input );
464                 p->y = getw( input );
465                 Push_Pixel( p, Buffer );
466                 Free_Pixel( p );
467             }
468             else
469                 while ( ret == 0 )
470                     continue;
471         } while ( ! feof( input ) );
472
473         fclose( input );
474     }
475
476     void Write_Pixel_Buffer( Buffer, fname )
477     char* Buffer;
478     {
479         XY_PIXEL* p;
480         FILE* out;
481
482         out = fopen( fname, "wb" );
483         if ( ! out )
484             return;
485
486         while ( ! feof( input ) )
487         {
488             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
489             if ( ! p )
490                 return;
491
492             ret = feof( input );
493             if ( ret == 0 )
494             {
495                 p->x = getw( input );
496                 p->y = getw( input );
497                 Push_Pixel( p, Buffer );
498                 Free_Pixel( p );
499             }
500             else
501                 while ( ret == 0 )
502                     continue;
503         } while ( ! feof( input ) );
504
505         fclose( input );
506     }
507
508     void Write_Pixel_Buffer( Buffer, fname )
509     char* Buffer;
510     {
511         XY_PIXEL* p;
512         FILE* out;
513
514         out = fopen( fname, "wb" );
515         if ( ! out )
516             return;
517
518         while ( ! feof( input ) )
519         {
520             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
521             if ( ! p )
522                 return;
523
524             ret = feof( input );
525             if ( ret == 0 )
526             {
527                 p->x = getw( input );
528                 p->y = getw( input );
529                 Push_Pixel( p, Buffer );
530                 Free_Pixel( p );
531             }
532             else
533                 while ( ret == 0 )
534                     continue;
535         } while ( ! feof( input ) );
536
537         fclose( input );
538     }
539
540     void Write_Pixel_Buffer( Buffer, fname )
541     char* Buffer;
542     {
543         XY_PIXEL* p;
544         FILE* out;
545
546         out = fopen( fname, "wb" );
547         if ( ! out )
548             return;
549
550         while ( ! feof( input ) )
551         {
552             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
553             if ( ! p )
554                 return;
555
556             ret = feof( input );
557             if ( ret == 0 )
558             {
559                 p->x = getw( input );
560                 p->y = getw( input );
561                 Push_Pixel( p, Buffer );
562                 Free_Pixel( p );
563             }
564             else
565                 while ( ret == 0 )
566                     continue;
567         } while ( ! feof( input ) );
568
569         fclose( input );
570     }
571
572     void Write_Pixel_Buffer( Buffer, fname )
573     char* Buffer;
574     {
575         XY_PIXEL* p;
576         FILE* out;
577
578         out = fopen( fname, "wb" );
579         if ( ! out )
580             return;
581
582         while ( ! feof( input ) )
583         {
584             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
585             if ( ! p )
586                 return;
587
588             ret = feof( input );
589             if ( ret == 0 )
590             {
591                 p->x = getw( input );
592                 p->y = getw( input );
593                 Push_Pixel( p, Buffer );
594                 Free_Pixel( p );
595             }
596             else
597                 while ( ret == 0 )
598                     continue;
599         } while ( ! feof( input ) );
600
601         fclose( input );
602     }
603
604     void Write_Pixel_Buffer( Buffer, fname )
605     char* Buffer;
606     {
607         XY_PIXEL* p;
608         FILE* out;
609
610         out = fopen( fname, "wb" );
611         if ( ! out )
612             return;
613
614         while ( ! feof( input ) )
615         {
616             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
617             if ( ! p )
618                 return;
619
620             ret = feof( input );
621             if ( ret == 0 )
622             {
623                 p->x = getw( input );
624                 p->y = getw( input );
625                 Push_Pixel( p, Buffer );
626                 Free_Pixel( p );
627             }
628             else
629                 while ( ret == 0 )
630                     continue;
631         } while ( ! feof( input ) );
632
633         fclose( input );
634     }
635
636     void Write_Pixel_Buffer( Buffer, fname )
637     char* Buffer;
638     {
639         XY_PIXEL* p;
640         FILE* out;
641
642         out = fopen( fname, "wb" );
643         if ( ! out )
644             return;
645
646         while ( ! feof( input ) )
647         {
648             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
649             if ( ! p )
650                 return;
651
652             ret = feof( input );
653             if ( ret == 0 )
654             {
655                 p->x = getw( input );
656                 p->y = getw( input );
657                 Push_Pixel( p, Buffer );
658                 Free_Pixel( p );
659             }
660             else
661                 while ( ret == 0 )
662                     continue;
663         } while ( ! feof( input ) );
664
665         fclose( input );
666     }
667
668     void Write_Pixel_Buffer( Buffer, fname )
669     char* Buffer;
670     {
671         XY_PIXEL* p;
672         FILE* out;
673
674         out = fopen( fname, "wb" );
675         if ( ! out )
676             return;
677
678         while ( ! feof( input ) )
679         {
680             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
681             if ( ! p )
682                 return;
683
684             ret = feof( input );
685             if ( ret == 0 )
686             {
687                 p->x = getw( input );
688                 p->y = getw( input );
689                 Push_Pixel( p, Buffer );
690                 Free_Pixel( p );
691             }
692             else
693                 while ( ret == 0 )
694                     continue;
695         } while ( ! feof( input ) );
696
697         fclose( input );
698     }
699
700     void Write_Pixel_Buffer( Buffer, fname )
701     char* Buffer;
702     {
703         XY_PIXEL* p;
704         FILE* out;
705
706         out = fopen( fname, "wb" );
707         if ( ! out )
708             return;
709
710         while ( ! feof( input ) )
711         {
712             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
713             if ( ! p )
714                 return;
715
716             ret = feof( input );
717             if ( ret == 0 )
718             {
719                 p->x = getw( input );
720                 p->y = getw( input );
721                 Push_Pixel( p, Buffer );
722                 Free_Pixel( p );
723             }
724             else
725                 while ( ret == 0 )
726                     continue;
727         } while ( ! feof( input ) );
728
729         fclose( input );
730     }
731
732     void Write_Pixel_Buffer( Buffer, fname )
733     char* Buffer;
734     {
735         XY_PIXEL* p;
736         FILE* out;
737
738         out = fopen( fname, "wb" );
739         if ( ! out )
740             return;
741
742         while ( ! feof( input ) )
743         {
744             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
745             if ( ! p )
746                 return;
747
748             ret = feof( input );
749             if ( ret == 0 )
750             {
751                 p->x = getw( input );
752                 p->y = getw( input );
753                 Push_Pixel( p, Buffer );
754                 Free_Pixel( p );
755             }
756             else
757                 while ( ret == 0 )
758                     continue;
759         } while ( ! feof( input ) );
760
761         fclose( input );
762     }
763
764     void Write_Pixel_Buffer( Buffer, fname )
765     char* Buffer;
766     {
767         XY_PIXEL* p;
768         FILE* out;
769
770         out = fopen( fname, "wb" );
771         if ( ! out )
772             return;
773
774         while ( ! feof( input ) )
775         {
776             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
777             if ( ! p )
778                 return;
779
780             ret = feof( input );
781             if ( ret == 0 )
782             {
783                 p->x = getw( input );
784                 p->y = getw( input );
785                 Push_Pixel( p, Buffer );
786                 Free_Pixel( p );
787             }
788             else
789                 while ( ret == 0 )
790                     continue;
791         } while ( ! feof( input ) );
792
793         fclose( input );
794     }
795
796     void Write_Pixel_Buffer( Buffer, fname )
797     char* Buffer;
798     {
799         XY_PIXEL* p;
800         FILE* out;
801
802         out = fopen( fname, "wb" );
803         if ( ! out )
804             return;
805
806         while ( ! feof( input ) )
807         {
808             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
809             if ( ! p )
810                 return;
811
812             ret = feof( input );
813             if ( ret == 0 )
814             {
815                 p->x = getw( input );
816                 p->y = getw( input );
817                 Push_Pixel( p, Buffer );
818                 Free_Pixel( p );
819             }
820             else
821                 while ( ret == 0 )
822                     continue;
823         } while ( ! feof( input ) );
824
825         fclose( input );
826     }
827
828     void Write_Pixel_Buffer( Buffer, fname )
829     char* Buffer;
830     {
831         XY_PIXEL* p;
832         FILE* out;
833
834         out = fopen( fname, "wb" );
835         if ( ! out )
836             return;
837
838         while ( ! feof( input ) )
839         {
840             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
841             if ( ! p )
842                 return;
843
844             ret = feof( input );
845             if ( ret == 0 )
846             {
847                 p->x = getw( input );
848                 p->y = getw( input );
849                 Push_Pixel( p, Buffer );
850                 Free_Pixel( p );
851             }
852             else
853                 while ( ret == 0 )
854                     continue;
855         } while ( ! feof( input ) );
856
857         fclose( input );
858     }
859
860     void Write_Pixel_Buffer( Buffer, fname )
861     char* Buffer;
862     {
863         XY_PIXEL* p;
864         FILE* out;
865
866         out = fopen( fname, "wb" );
867         if ( ! out )
868             return;
869
870         while ( ! feof( input ) )
871         {
872             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
873             if ( ! p )
874                 return;
875
876             ret = feof( input );
877             if ( ret == 0 )
878             {
879                 p->x = getw( input );
880                 p->y = getw( input );
881                 Push_Pixel( p, Buffer );
882                 Free_Pixel( p );
883             }
884             else
885                 while ( ret == 0 )
886                     continue;
887         } while ( ! feof( input ) );
888
889         fclose( input );
890     }
891
892     void Write_Pixel_Buffer( Buffer, fname )
893     char* Buffer;
894     {
895         XY_PIXEL* p;
896         FILE* out;
897
898         out = fopen( fname, "wb" );
899         if ( ! out )
900             return;
901
902         while ( ! feof( input ) )
903         {
904             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
905             if ( ! p )
906                 return;
907
908             ret = feof( input );
909             if ( ret == 0 )
910             {
911                 p->x = getw( input );
912                 p->y = getw( input );
913                 Push_Pixel( p, Buffer );
914                 Free_Pixel( p );
915             }
916             else
917                 while ( ret == 0 )
918                     continue;
919         } while ( ! feof( input ) );
920
921         fclose( input );
922     }
923
924     void Write_Pixel_Buffer( Buffer, fname )
925     char* Buffer;
926     {
927         XY_PIXEL* p;
928         FILE* out;
929
930         out = fopen( fname, "wb" );
931         if ( ! out )
932             return;
933
934         while ( ! feof( input ) )
935         {
936             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
937             if ( ! p )
938                 return;
939
940             ret = feof( input );
941             if ( ret == 0 )
942             {
943                 p->x = getw( input );
944                 p->y = getw( input );
945                 Push_Pixel( p, Buffer );
946                 Free_Pixel( p );
947             }
948             else
949                 while ( ret == 0 )
950                     continue;
951         } while ( ! feof( input ) );
952
953         fclose( input );
954     }
955
956     void Write_Pixel_Buffer( Buffer, fname )
957     char* Buffer;
958     {
959         XY_PIXEL* p;
960         FILE* out;
961
962         out = fopen( fname, "wb" );
963         if ( ! out )
964             return;
965
966         while ( ! feof( input ) )
967         {
968             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
969             if ( ! p )
970                 return;
971
972             ret = feof( input );
973             if ( ret == 0 )
974             {
975                 p->x = getw( input );
976                 p->y = getw( input );
977                 Push_Pixel( p, Buffer );
978                 Free_Pixel( p );
979             }
980             else
981                 while ( ret == 0 )
982                     continue;
983         } while ( ! feof( input ) );
984
985         fclose( input );
986     }
987
988     void Write_Pixel_Buffer( Buffer, fname )
989     char* Buffer;
990     {
991         XY_PIXEL* p;
992         FILE* out;
993
994         out = fopen( fname, "wb" );
995         if ( ! out )
996             return;
997
998         while ( ! feof( input ) )
999         {
1000             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1001             if ( ! p )
1002                 return;
1003
1004             ret = feof( input );
1005             if ( ret == 0 )
1006             {
1007                 p->x = getw( input );
1008                 p->y = getw( input );
1009                 Push_Pixel( p, Buffer );
1010                 Free_Pixel( p );
1011             }
1012             else
1013                 while ( ret == 0 )
1014                     continue;
1015         } while ( ! feof( input ) );
1016
1017         fclose( input );
1018     }
1019
1020     void Write_Pixel_Buffer( Buffer, fname )
1021     char* Buffer;
1022     {
1023         XY_PIXEL* p;
1024         FILE* out;
1025
1026         out = fopen( fname, "wb" );
1027         if ( ! out )
1028             return;
1029
1030         while ( ! feof( input ) )
1031         {
1032             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1033             if ( ! p )
1034                 return;
1035
1036             ret = feof( input );
1037             if ( ret == 0 )
1038             {
1039                 p->x = getw( input );
1040                 p->y = getw( input );
1041                 Push_Pixel( p, Buffer );
1042                 Free_Pixel( p );
1043             }
1044             else
1045                 while ( ret == 0 )
1046                     continue;
1047         } while ( ! feof( input ) );
1048
1049         fclose( input );
1050     }
1051
1052     void Write_Pixel_Buffer( Buffer, fname )
1053     char* Buffer;
1054     {
1055         XY_PIXEL* p;
1056         FILE* out;
1057
1058         out = fopen( fname, "wb" );
1059         if ( ! out )
1060             return;
1061
1062         while ( ! feof( input ) )
1063         {
1064             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1065             if ( ! p )
1066                 return;
1067
1068             ret = feof( input );
1069             if ( ret == 0 )
1070             {
1071                 p->x = getw( input );
1072                 p->y = getw( input );
1073                 Push_Pixel( p, Buffer );
1074                 Free_Pixel( p );
1075             }
1076             else
1077                 while ( ret == 0 )
1078                     continue;
1079         } while ( ! feof( input ) );
1080
1081         fclose( input );
1082     }
1083
1084     void Write_Pixel_Buffer( Buffer, fname )
1085     char* Buffer;
1086     {
1087         XY_PIXEL* p;
1088         FILE* out;
1089
1090         out = fopen( fname, "wb" );
1091         if ( ! out )
1092             return;
1093
1094         while ( ! feof( input ) )
1095         {
1096             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1097             if ( ! p )
1098                 return;
1099
1100             ret = feof( input );
1101             if ( ret == 0 )
1102             {
1103                 p->x = getw( input );
1104                 p->y = getw( input );
1105                 Push_Pixel( p, Buffer );
1106                 Free_Pixel( p );
1107             }
1108             else
1109                 while ( ret == 0 )
1110                     continue;
1111         } while ( ! feof( input ) );
1112
1113         fclose( input );
1114     }
1115
1116     void Write_Pixel_Buffer( Buffer, fname )
1117     char* Buffer;
1118     {
1119         XY_PIXEL* p;
1120         FILE* out;
1121
1122         out = fopen( fname, "wb" );
1123         if ( ! out )
1124             return;
1125
1126         while ( ! feof( input ) )
1127         {
1128             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1129             if ( ! p )
1130                 return;
1131
1132             ret = feof( input );
1133             if ( ret == 0 )
1134             {
1135                 p->x = getw( input );
1136                 p->y = getw( input );
1137                 Push_Pixel( p, Buffer );
1138                 Free_Pixel( p );
1139             }
1140             else
1141                 while ( ret == 0 )
1142                     continue;
1143         } while ( ! feof( input ) );
1144
1145         fclose( input );
1146     }
1147
1148     void Write_Pixel_Buffer( Buffer, fname )
1149     char* Buffer;
1150     {
1151         XY_PIXEL* p;
1152         FILE* out;
1153
1154         out = fopen( fname, "wb" );
1155         if ( ! out )
1156             return;
1157
1158         while ( ! feof( input ) )
1159         {
1160             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1161             if ( ! p )
1162                 return;
1163
1164             ret = feof( input );
1165             if ( ret == 0 )
1166             {
1167                 p->x = getw( input );
1168                 p->y = getw( input );
1169                 Push_Pixel( p, Buffer );
1170                 Free_Pixel( p );
1171             }
1172             else
1173                 while ( ret == 0 )
1174                     continue;
1175         } while ( ! feof( input ) );
1176
1177         fclose( input );
1178     }
1179
1180     void Write_Pixel_Buffer( Buffer, fname )
1181     char* Buffer;
1182     {
1183         XY_PIXEL* p;
1184         FILE* out;
1185
1186         out = fopen( fname, "wb" );
1187         if ( ! out )
1188             return;
1189
1190         while ( ! feof( input ) )
1191         {
1192             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1193             if ( ! p )
1194                 return;
1195
1196             ret = feof( input );
1197             if ( ret == 0 )
1198             {
1199                 p->x = getw( input );
1200                 p->y = getw( input );
1201                 Push_Pixel( p, Buffer );
1202                 Free_Pixel( p );
1203             }
1204             else
1205                 while ( ret == 0 )
1206                     continue;
1207         } while ( ! feof( input ) );
1208
1209         fclose( input );
1210     }
1211
1212     void Write_Pixel_Buffer( Buffer, fname )
1213     char* Buffer;
1214     {
1215         XY_PIXEL* p;
1216         FILE* out;
1217
1218         out = fopen( fname, "wb" );
1219         if ( ! out )
1220             return;
1221
1222         while ( ! feof( input ) )
1223         {
1224             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1225             if ( ! p )
1226                 return;
1227
1228             ret = feof( input );
1229             if ( ret == 0 )
1230             {
1231                 p->x = getw( input );
1232                 p->y = getw( input );
1233                 Push_Pixel( p, Buffer );
1234                 Free_Pixel( p );
1235             }
1236             else
1237                 while ( ret == 0 )
1238                     continue;
1239         } while ( ! feof( input ) );
1240
1241         fclose( input );
1242     }
1243
1244     void Write_Pixel_Buffer( Buffer, fname )
1245     char* Buffer;
1246     {
1247         XY_PIXEL* p;
1248         FILE* out;
1249
1250         out = fopen( fname, "wb" );
1251         if ( ! out )
1252             return;
1253
1254         while ( ! feof( input ) )
1255         {
1256             p = (XY_PIXEL*) malloc( sizeof( XY_PIXEL ) );
1257             if ( ! p )
1258                 return;
1259
1260             ret = feof( input );
1261             if ( ret == 0 )
1262             {
1263                 p->x = getw( input );
1264                 p->y = getw( input );
1265                 Push_Pixel( p, Buffer );
1266                 Free_Pixel( p );
1267             }
1268             else
1269                 while ( ret == 0 )
1270                     continue;
1271         } while ( ! feof( input ) );
1272
1273         fclose( input );
1274     }
1275
1276     void Write_Pixel_Buffer( Buffer, fname )
1277     char* Buffer;
1278     {
1279         XY_PIXEL* p;
1280         FILE* out;
1281
1282         out = fopen( fname,
```

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  #include <stdio.h>
6  #include <math.h>
7  #include <graphics.h>
8  #include <dos.h>
9  #include <conio.h>
10 #include <values.h>
11 #include <string.h>
12 #include <assert.h>
13 #include "popenus.h"
14 #include "memory.h"
15 #include "pivmenus.h"
16
17 MENU Main;
18 MENU File;
19 MENU File_Plv;
20 MENU Piv_Scaler;
21
22 void Define_Menus()
23 {
24     /* Define menu title and option texts */
25
26     F2_NAME = "Save Particles"; F2_DEF = "Write PIV Data";
27     F3_NAME = "Open PIV Files"; F3_DEF = "Open TIF File";
28     F4_NAME = "Dos Command"; F4_DEF = "System Command";
29
30     Main.title = "Tiff Utility";
31
32     Main.op[0] = "File TIF";
33     Main.op[1] = "File PIV";
34     Main.op[2] = "View TIF";
35     Main.op[3] = "View Vec";
36     Main.op[4] = "Zoom In";
37     Main.op[5] = "Filter";
38     Main.op[6] = "Process";
39     Main.op[7] = "Batch";
40     Main.op[8] = "END";
41
42     /* Define help for each option */
43
44     /* Define functions or menus called by options */
45
46     Main.op[0] = &File; Main.op[1] = &CALLS_MENU;
47     Main.op[2] = &File_Plv; Main.op[3] = &CALLS_MENU;
48     Main.op[4] = &Display_Plv; Main.op[5] = &CALLS_FUNC;
49     Main.op[6] = &Display_Plv_Particles_Zoomed_Out; Main.op[7] = &CALLS_FUNC;
50     Main.op[8] = &Display_Plv_Particles_Palred; Main.op[9] = &CALLS_FUNC;
51     Main.op[10] = &Plv_Scaler; Main.op[11] = &CALLS_FUNC;
52     Main.op[12] = &Process_Plv; Main.op[13] = &CALLS_FUNC;
53     Main.op[14] = &Batch_Process; Main.op[15] = &CALLS_FUNC;
54
55     /* Define option that selection bar is to appear over first */
56
57     Main.last_selection = 0;
58
59     /* Define menu title and option texts */
60
61     File.title = "File";
62
63     File.op[0] = "Open PIV Files F3";
64     File.op[1] = "Tif Information";
65     File.op[2] = "Exit";
66

```

```

67 File.op[3] = "END";
68
69 /* Define help for each option */
70
71 /* Define functions or menus called by options */
72 File.op[0] = &Open_Tif_File; File.op[1] = &CALLS_FUNC;
73 File.op[2] = &Print_Tif_Dir; File.op[3] = &CALLS_FUNC;
74 File.op[4] = &Quit; File.op[5] = &CALLS_FUNC;
75
76 File.op[12] = &Quit; File.op[13] = &CALLS_FUNC;
77
78 /* Define option that selection bar is to appear over first */
79 File.last_selection = 0;
80
81 Piv_Scaler.title = "Piv Parameters";
82
83 /* Define menu title and option texts */
84
85 Piv_Scaler.op[0] = "Scale ( 1 cm in scanned image == 2 cm in PIV experiment )";
86
87 Piv_Scaler.op[1] = "Threshold Intensity level for 8 bit TIF";
88 Piv_Scaler.op[2] = "Minimum particle size square pixels ( TIF )";
89 Piv_Scaler.op[3] = "Maximum particle size square pixels ( TIF )";
90 Piv_Scaler.op[4] = "Minimum feature size square pixels ( TIF )";
91 Piv_Scaler.op[5] = "Pulse separation ( micro seconds )";
92 Piv_Scaler.op[6] = "Minimum velocity accepted ( metres/second )";
93 Piv_Scaler.op[7] = "Maximum velocity accepted ( metres/second )";
94 Piv_Scaler.op[8] = "Minimum angle accepted ( 0-north, 90-east etc )";
95 Piv_Scaler.op[9] = "Maximum angle accepted ( 0-north, 90-east etc )";
96 Piv_Scaler.op[10] = "Set Box size in microns for direction validation";
97 Piv_Scaler.op[11] = "Set vector plot scaling factor";
98 Piv_Scaler.op[12] = "END";
99
100 /* Define help for each option */
101
102 Piv_Scaler.op[0] = "Scale ( 1 cm in scanned image == 2 cm in PIV experiment )";
103
104 Piv_Scaler.op[1] = "Threshold Intensity level for 8 bit TIF";
105 Piv_Scaler.op[2] = "Minimum particle size square pixels ( TIF )";
106 Piv_Scaler.op[3] = "Maximum particle size square pixels ( TIF )";
107 Piv_Scaler.op[4] = "Minimum feature size square pixels ( TIF )";
108 Piv_Scaler.op[5] = "Pulse separation ( micro seconds )";
109 Piv_Scaler.op[6] = "Minimum velocity accepted ( metres/second )";
110 Piv_Scaler.op[7] = "Maximum velocity accepted ( metres/second )";
111 Piv_Scaler.op[8] = "Minimum angle accepted ( 0-north, 90-east etc )";
112 Piv_Scaler.op[9] = "Maximum angle accepted ( 0-north, 90-east etc )";
113 Piv_Scaler.op[10] = "Set Box size in microns for direction validation";
114 Piv_Scaler.op[11] = "Set vector plot scaling factor";
115
116 /* Define functions or menus called by options */
117
118 Piv_Scaler.op[0] = &Set_Scale_For_1cm_In_Scanned_Image; Piv_Scaler.op[1] = &CALLS_FUNC;
119 Piv_Scaler.op[1] = &KEEP_MENU; Piv_Scaler.op[2] = &CALLS_FUNC;
120 Piv_Scaler.op[2] = &Set_Min_Pulse; Piv_Scaler.op[3] = &CALLS_FUNC;
121 Piv_Scaler.op[3] = &Set_Max_Pulse; Piv_Scaler.op[4] = &CALLS_FUNC;
122 Piv_Scaler.op[4] = &Set_Min_Feature_Size; Piv_Scaler.op[5] = &CALLS_FUNC;
123 Piv_Scaler.op[5] = &Set_Pulse_Separation; Piv_Scaler.op[6] = &CALLS_FUNC;
124 Piv_Scaler.op[6] = &Min_Vel; Piv_Scaler.op[7] = &CALLS_FUNC;
125 Piv_Scaler.op[8] = &KEEP_MENU; Piv_Scaler.op[9] = &CALLS_FUNC;
126 Piv_Scaler.op[10] = &KEEP_MENU; Piv_Scaler.op[11] = &CALLS_FUNC;
127 Piv_Scaler.op[12] = &KEEP_MENU; Piv_Scaler.op[13] = &CALLS_FUNC;
128

```

```

124 Piv_Scale.op[7] = Max_Vel; Piv_Scale.tp[7] = CALLS_FUNC |
KEEP_MENU;
125 Piv_Scale.op[8] = Min_Dir; Piv_Scale.tp[8] = CALLS_FUNC |
KEEP_MENU;
126 Piv_Scale.op[9] = Max_Dir; Piv_Scale.tp[9] = CALLS_FUNC |
KEEP_MENU;
127 Piv_Scale.op[10] = Set_Box_Size; Piv_Scale.tp[10] = CALLS_FUNC |
KEEP_MENU;
128 Piv_Scale.op[11] = Set_Vec_Scale; Piv_Scale.tp[11] = CALLS_FUNC |
KEEP_MENU;
129
130 /* Define option that selection bar is to appear over first */
131 Piv_Scale.last_selection = 0;
132
133 File_piv.title = "Piv File Menu";
134
135 /* Define menu title and option texts */
136
137 File_piv.op[0] = "Read all data files with current prefix";
138 File_piv.op[1] = "Write AP configuration file [.cfg]";
139 File_piv.op[2] = "Read AP configuration file [.cfg]";
140 File_piv.op[3] = "Write textual data on particles [.txt]";
141 File_piv.op[4] = "Write pair velocity vector table [.vec]";
142 File_piv.op[5] = "Write pair velocity vector table [.par]";
143 File_piv.op[6] = "Read binary data on particles";
144 File_piv.op[7] = "Write Matlab plot file";
145 File_piv.op[8] = "END";
146
147
148 /* Define help for each option */
149
150 File_piv.op[0] = "";
151 File_piv.op[1] = "";
152 File_piv.op[2] = "";
153 File_piv.op[3] = "";
154 File_piv.op[4] = "";
155 File_piv.op[5] = "";
156 File_piv.op[6] = "";
157 File_piv.op[7] = "";
158
159 /* Define functions or menus called by options */
160
161 File_piv.op[0] = Read_All_Piv_Files; File_piv.tp[0] = CALLS_FUNC
162 NC | KEEP_MENU;
163 File_piv.op[1] = Write_Piv_Parameter_Data_File; File_piv.tp[1] = CALLS_FUNC
164 NC | KEEP_MENU;
165 File_piv.op[2] = Read_Piv_Parameter_Data_File; File_piv.tp[2] = CALLS_FUNC
166 NC | KEEP_MENU;
167 File_piv.op[3] = Write_Particle_Buffer_Text; File_piv.tp[3] = CALLS_FUNC
168 NC | KEEP_MENU;
169 File_piv.op[4] = Write_Particle_Pair_Velocities; File_piv.tp[4] = CALLS_FUNC
170 NC | KEEP_MENU;
171 File_piv.op[5] = Write_Piv_Data; File_piv.tp[5] = CALLS_FUNC
172 NC | KEEP_MENU;
173 File_piv.op[6] = Read_Piv_Data_Par; File_piv.tp[6] = CALLS_FUNC
174 NC | KEEP_MENU;
175 File_piv.op[7] = Write_Plot_File; File_piv.tp[7] = CALLS_FUNC
176 NC | KEEP_MENU;
177
178 /* Define option that selection bar is to appear over first */
179 File_piv.last_selection = 0;
180
181 }

```

```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5
6  typedef struct xy_particle
7  {
8      int xp,yp;
9      long mass;
10     char deleter;
11     struct xy_particle huge* pred;
12     struct xy_particle huge* succ;
13 } XY_PARTICLE;
14
15 typedef XY_PARTICLE huge* PARTICLE_POINTER;
16
17 typedef struct partindex
18 {
19     struct xy_particle huge* p; long n; } PARTINDEX;
20
21 typedef struct particle_buffer
22 {
23     struct xy_particle huge* head;
24     struct partindex cparticle;
25     struct partindex lparticle; } PBUFF;
26
27 PARTICLE_POINTER Init_Particle_Buffer();
28 PARTICLE_POINTER Make_Particle();
29 PARTICLE_POINTER Insert_Particle();
30 PARTICLE_POINTER Del_Particle();
31 PARTICLE_POINTER Pop_Particle();
32 PARTICLE_POINTER Set_Curr_Particle();
33 void Push_Particle();
34 void Free_Particle();
35 void Delete_All_Particles();
36 void Delete_Marked_Particles();
37 void Swap_Particles();
38 long Curr_Particle();
39 Last_Particle();
40 extern PBUFF Particles;
```

```

1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  #include <stdarg.h>
6  #include <stdio.h>
7  #include <alloc.h>
8  #include <math.h>
9  #include "plist.h"
10 #include "memory.h"
11
12 #ifdef PARTICLES
13
14     PARTICLE_POINTER Init_Particle_Buffer()
15     {
16         PARTICLES.head = ( PARTICLE_POINTER ) Malloc( (unsigned long) sizeof( XY_PARTI
17         CLE ) );
18     }
19     if ( PARTICLES.head != NULL )
20     {
21         PARTICLES.head->succ = PARTICLES.head;
22         PARTICLES.head->pred = PARTICLES.head;
23         PARTICLES.iparticle.n = 0;
24         PARTICLES.iparticle.p = PARTICLES.head;
25         PARTICLES.cparticle = PARTICLES.iparticle;
26     }
27     return( PARTICLES.head );
28 }
29
30 PARTICLE_POINTER Make_Particle( x, y, mass )
31 int x,y
32 long mass;
33 {
34     PARTICLE_POINTER p;
35
36     p = ( PARTICLE_POINTER ) Malloc( (unsigned long) sizeof( XY_PARTICLE ) );
37
38     if ( p == NULL )
39     {
40         printf("Malloc failed in particle memory allocation\n");
41         exit(1);
42     }
43
44     p->xp = x;
45     p->yp = y;
46     p->mass = mass;
47     p->delete = (char) 0;
48
49     return( p );
50 }
51
52 void Push_Particle( particle )
53 PARTICLE_POINTER particle;
54 {
55     PARTICLE_POINTER p;
56     long n;
57
58     if ( particle != NULL )
59     {
60         p = PARTICLES.iparticle.p;
61         n = PARTICLES.iparticle.n;
62         particle->succ = p->succ;
63         particle->pred = particle;
64         particle->pred = p;
65     }

```

```

66     p->succ = particle;
67
68     PARTICLES.iparticle.p = PARTICLES.head->pred;
69     PARTICLES.iparticle.n++;
70
71     PARTICLES.cparticle.p = particle;
72     PARTICLES.cparticle.n = ++n;
73 }
74
75 PARTICLE_POINTER Pop_Particle()
76 {
77     PARTICLE_POINTER p;
78     PARTICLE_POINTER pl;
79     PARTICLE_POINTER pr;
80
81     pr = PARTICLES.cparticle.p;
82
83     if ( pr != PARTICLES.head )
84     {
85         p = pr->pred;
86
87         PARTICLES.cparticle.p = p;
88         PARTICLES.cparticle.n--;
89
90         pl = p->succ->succ;
91
92         p->succ = pl;
93         pl->pred = p;
94
95         PARTICLES.iparticle.n--;
96         PARTICLES.iparticle.p = PARTICLES.head->pred;
97     }
98     else pr = NULL;
99
100     return( pr );
101 }
102
103 PARTICLE_POINTER Insert_Particle(n,particle)
104 long n;
105 PARTICLE_POINTER particle;
106 {
107     PARTICLE_POINTER p;
108
109     if ( particle != NULL )
110     {
111         p = Set_Curr_Particle(n);
112         particle->succ = p->succ;
113         p->succ->pred = particle;
114         particle->pred = p;
115         p->succ = particle;
116
117         PARTICLES.iparticle.p = PARTICLES.head->pred;
118         PARTICLES.iparticle.n++;
119         PARTICLES.cparticle.p = particle;
120         PARTICLES.cparticle.n = ++n;
121     }
122     return( particle );
123 }
124
125 void Free_Particle( particle )
126 PARTICLE_POINTER particle;
127 {
128     if ( particle != NULL ) Free( particle );
129 }
130
131

```



```

132 }
133 PARTICLE_POINTER Del_Particle(n)
134 {
135     long n;
136     PARTICLE_POINTER p;
137     PARTICLE_POINTER pl;
138     p = NULL;
139     if ( n > 0 )
140     {
141         p = Set_Curr_Particle(n-1);
142         pl = p->succ->succ;
143         Free_Particle( p->succ );
144         p->succ = pl;
145         pl->pred = p;
146         Particles.iParticle.n -= 1;
147         Particles.iParticle.p = Particles.head->pred;
148         return( NULL );
149     }
150     PARTICLE_POINTER Set_Curr_Particle(n)
151     {
152         long n;
153         long nstep;
154         long diff;
155         PARTICLE_POINTER p = Particles.head;
156         if ( n > Particles.iParticle.n ) n = Particles.iParticle.n;
157         nstep = n;
158         diff = n - Particles.iParticle.n;
159         if ( labs(diff) < nstep ) { p = Particles.iParticle.p; nstep = diff; }
160         if ( labs(diff) < labs(nstep) ) { p = Particles.iParticle.p; nstep = diff; }
161         else { while( nstep-- ) p=p->succ; }
162         Particles.iParticle.n = n;
163         Particles.iParticle.p = p;
164         return( p );
165     }
166     void Delete_All_Particles()
167     {
168         PARTICLE_POINTER p;
169         PARTICLE_POINTER pl;
170         if ( Particles.head != NULL )
171         {
172             p = Particles.head->succ;
173             while( p != Particles.head )
174             {
175                 pl = p->succ;
176                 Free_Particle( p );
177                 p = pl;
178             }
179             Particles.head->succ = Particles.head;
180         }
181     }

```

```

198 Particles.head->pred = Particles.head;
199 Particles.iParticle.n = 0;
200 Particles.iParticle.p = Particles.head;
201 Particles.iParticle = Particles.iParticle;
202 }
203 void Delete_Marked_Particles()
204 {
205     PARTICLE_POINTER p;
206     PARTICLE_POINTER pl;
207     if ( Particles.head != NULL )
208     {
209         p = Particles.head->succ;
210         while( p != Particles.head )
211         {
212             pl = p->succ;
213             if ( p->delete == (char) 1 )
214             {
215                 Free_Particle( p->succ );
216                 p->succ = pl;
217                 pl->pred = p;
218                 Particles.iParticle.n -= 1;
219             }
220             p = pl;
221         }
222     }
223     void Swap_Particles( a, b )
224     {
225         PARTICLE_POINTER a;
226         PARTICLE_POINTER b;
227         XY_PARTICLE t;
228         if ( a != NULL && b != NULL ) {
229             t = *a;
230             a->xp = b->xp;
231             a->yp = b->yp;
232             a->mass = b->mass;
233             a->delete = b->delete;
234             b->xp = t.xp;
235             b->yp = t.yp;
236             b->mass = t.mass;
237             b->delete = t.delete;
238         }
239     }
240     long Curr_Particle()
241     {
242         return( Particles.iParticle );
243     }
244     long Last_Particle()
245     {
246         return( Particles.iParticle );
247     }

```

```

1  /* Copyright (c) 1991 by Tom Judge.
2  * All rights reserved.
3
4  */
5  #include <stdarg.h>
6  #include <stdio.h>
7  #include <graphics.h>
8  #include <dos.h>
9  #include <ctype.h>
10 #include "popmenus.h"
11 #include "memory.h"
12
13
14
15 unsigned char model8_pal_tab_min_is_black[] =
16 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17   17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
18   33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
19   49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63 };
20
21
22 unsigned char model8_pal_tab_min_is_white[] =
23 { 63, 63, 63, 63, 58, 58, 58, 53, 53, 53, 48, 48, 48, 48,
24   44, 44, 44, 40, 40, 40, 36, 36, 36, 32, 32, 32, 32,
25   28, 28, 28, 24, 24, 24, 20, 20, 20, 16, 16, 16,
26   12, 12, 12, 8, 8, 8, 8, 4, 4, 4, 0, 0, 0 };
27
28 unsigned char model8_pointers[] =
29 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0 };
30
31
32 char pop_line[ 512 ];
33 int last_opt_x;
34 int last_opt_y;
35
36 int bar_colour;
37
38 MENU* current_menu;
39 int current_options;
40
41 #define INPUT_LENGTH 30
42
43 int top_print;
44 int bottom_print;
45
46 PF pop_fkey[ 9 ];
47 char* pop_fkey_name[ 9 ];
48
49 int main_menu_flag = 0;
50 int main_menu_key = 0;
51
52 byte attribute, character;
53
54 #define VIDINT 0x10 /* INT 10H video calls */
55 #define CIS 0x07 /* system call - 07H - Initialize window */
56 #define SETCUR 0x02 /* system call - 02H - Set cursor position */
57 #define WRATRCR 0x09 /* system call - 09H - Write attribute and
58   character at cursor */
59 #define RDATRCR 0x08 /* system call - 08H - Read attribute and
60   character at cursor */
61 #define SERVID 0x00 /* system call - 00H - Set video mode */
62 #define GETVID 0x00 /* system call - 07H - Get video mode */
63 #define SETDPC 0x03 /* system call - 03H - Set display page */
64 #define T0031B 0x10 /* system call - 10H - toggle Blink/intensity Bit */
65
66

```

```

67 union REGS regs;
68 void Set_Block_Palette( min_is_black )
69 {
70   int min_is_black;
71   unsigned int temp;
72   unsigned char* palette;
73   if ( min_is_black ) palette = model8_pal_tab_min_is_black;
74   else palette = model8_pal_tab_min_is_white;
75
76   regs.h.ah = 0x10; /* AH is system call number */
77   regs.h.al = 0x02;
78   regs.x.dx = 0; /* offset of model8_pointers */
79   temp = ES; ES = 0; /* segment of model8_pointers */
80   int86( VIDINT, &regs, &regs );
81   _ES = temp;
82
83   regs.h.ah = 0x10; /* AH is system call number */
84   regs.h.al = 0x12;
85   regs.x.cx = 0;
86   regs.x.dx = 16;
87   regs.x.cx = 16;
88   temp = ES; ES = 0; /* offset of palette */
89   int86( VIDINT, &regs, &regs );
90   _ES = temp;
91
92   void Set_Video_Mode()
93   {
94     regs.h.ah = SERVID; /* AH is system call number */
95     regs.h.al = 0x03;
96     int86( VIDINT, &regs, &regs );
97
98     void Set_Text_Page( page )
99     {
100       regs.h.ah = SETDPC;
101       regs.h.al = 0;
102       int86( VIDINT, &regs, &regs );
103     }
104
105     void Set_Page()
106     {
107       regs.h.ah = SETDPC;
108       regs.h.al = page;
109       int86( VIDINT, &regs, &regs );
110     }
111
112     void Set_Repeat_Rate()
113     {
114       regs.h.ah = 0x3;
115       regs.h.al = 0x5;
116       regs.h.bh = 0x3;
117       regs.h.bl = 0x0;
118       int86( 0x16, &regs, &regs );
119     }
120
121     void Get_Char_Attr()
122     {
123       regs.h.ah = RDATRCR; /* AH is system call number */
124       regs.h.bh = 0; /* page 0 */
125     }
126
127
128
129
130
131
132

```

```

133  int86( VIDINT, &regs, &regs);
134  character = regs.h.al;
135  attribute = regs.h.ah;
136  }
137  }
138  }
139  void Write_Char_Attr( character, attribute, replicate )
140  byte character, attribute, replicate;
141  {
142  regs.h.ah = WRATCRH; /* AH is system call number */
143  regs.h.bh = 0; /* page 0 */
144  regs.h.al = character;
145  regs.h.bl = attribute;
146  regs.h.cl = replicate;
147  regs.h.ch = 0;
148  int86( VIDINT, &regs, &regs);
149  }
150  }
151  void Set_Text_Pos( x,y )
152  int x,y;
153  {
154  regs.h.ah = SETCUR; /* AH is system call number */
155  regs.h.bh = 0; /* page 0 */
156  regs.h.ch = y;
157  regs.h.dl = x;
158  int86( VIDINT, &regs, &regs);
159  }
160  }
161  void Bye_Cursor()
162  {
163  regs.h.ah = SETCUR; /* AH is system call number */
164  regs.h.bh = 0; /* page 0 */
165  regs.h.ch = 30;
166  regs.h.dl = 0;
167  int86( VIDINT, &regs, &regs);
168  }
169  }
170  int Get_Video_Mode()
171  {
172  int Get_Video_Mode();
173  regs.h.ah = GETVID; /* AH is system call number */
174  int86( VIDINT, &regs, &regs);
175  return( (int) regs.h.al );
176  }
177  }
178  }
179  }
180  }
181  int Get_Text_X_Max( video_mode )
182  int video_mode;
183  {
184  int x;
185  switch( video_mode ) {
186  case 0x0 : x = 39; break;
187  case 0x1 : x = 39; break;
188  case 0x2 : x = 39; break;
189  case 0x3 : x = 39; break;
190  case 0x4 : x = 39; break;
191  case 0x5 : x = 39; break;
192  case 0x6 : x = 39; break;
193  case 0x7 : x = 39; break;
194  case 0x8 : x = 39; break;
195  case 0x9 : x = 39; break;
196  case 0xa : x = 39; break;
197  case 0xb : x = 39; break;
198  case 0xc : x = 39; break;

```

```

199  case 0xa : x = 39; break;
200  case 0xb : x = 39; break;
201  case 0xc : x = 39; break;
202  case 0xd : x = 39; break;
203  case 0xe : x = 39; break;
204  case 0xf : x = 39; break;
205  case 0x10 : x = 39; break;
206  case 0x11 : x = 39; break;
207  case 0x12 : x = 39; break;
208  case 0x13 : x = 39; break;
209  }
210  return( x );
211  }
212  }
213  int Get_Text_Y_Max( video_mode )
214  int video_mode;
215  {
216  int y;
217  switch( video_mode ) {
218  case 0x0 : y = 24; break;
219  case 0x1 : y = 24; break;
220  case 0x2 : y = 24; break;
221  case 0x3 : y = 24; break;
222  case 0x4 : y = 24; break;
223  case 0x5 : y = 24; break;
224  case 0x6 : y = 24; break;
225  case 0x7 : y = 24; break;
226  case 0x8 : y = 24; break;
227  case 0x9 : y = 24; break;
228  case 0xa : y = 24; break;
229  case 0xb : y = 24; break;
230  case 0xc : y = 24; break;
231  case 0xd : y = 24; break;
232  case 0xe : y = 24; break;
233  case 0xf : y = 24; break;
234  case 0x10 : y = 24; break;
235  case 0x11 : y = 24; break;
236  case 0x12 : y = 24; break;
237  case 0x13 : y = 24; break;
238  }
239  return( y );
240  }
241  }
242  }
243  }
244  }
245  }
246  }
247  void CIs()
248  {
249  regs.h.ah = CIs; /* AH is system call number */
250  regs.h.al = 0; /* if AL = zero entire window is blanked */
251  regs.h.bh = 7; /* Attribute, normal video */
252  regs.h.ch = 0; /* y coordinate of upper left corner */
253  regs.h.cl = 0; /* x coordinate of upper left corner */
254  regs.h.dh = 24; /* y coordinate of lower right corner */
255  regs.h.dl = 39; /* x coordinate of lower right corner */
256  int86( VIDINT, &regs, &regs);
257  }
258  }
259  }
260  }
261  }
262  }
263  }
264  }
265  }
266  }
267  }
268  }
269  }
270  }
271  }
272  }
273  }
274  }
275  }
276  }
277  }
278  }
279  }
280  }
281  }
282  }
283  }
284  }
285  }
286  }
287  }
288  }
289  }
290  }
291  }
292  }
293  }
294  }
295  }
296  }
297  }
298  }
299  }
300  }
301  }
302  }
303  }
304  }
305  }
306  }
307  }
308  }
309  }
310  }
311  }
312  }
313  }
314  }
315  }
316  }
317  }
318  }
319  }
320  }
321  }
322  }
323  }
324  }
325  }
326  }
327  }
328  }
329  }
330  }
331  }
332  }
333  }
334  }
335  }
336  }
337  }
338  }
339  }
340  }
341  }
342  }
343  }
344  }
345  }
346  }
347  }
348  }
349  }
350  }
351  }
352  }
353  }
354  }
355  }
356  }
357  }
358  }
359  }
360  }
361  }
362  }
363  }
364  }
365  }
366  }
367  }
368  }
369  }
370  }
371  }
372  }
373  }
374  }
375  }
376  }
377  }
378  }
379  }
380  }
381  }
382  }
383  }
384  }
385  }
386  }
387  }
388  }
389  }
390  }
391  }
392  }
393  }
394  }
395  }
396  }
397  }
398  }
399  }
400  }
401  }
402  }
403  }
404  }
405  }
406  }
407  }
408  }
409  }
410  }
411  }
412  }
413  }
414  }
415  }
416  }
417  }
418  }
419  }
420  }
421  }
422  }
423  }
424  }
425  }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }
481  }
482  }
483  }
484  }
485  }
486  }
487  }
488  }
489  }
490  }
491  }
492  }
493  }
494  }
495  }
496  }
497  }
498  }
499  }
500  }
501  }
502  }
503  }
504  }
505  }
506  }
507  }
508  }
509  }
510  }
511  }
512  }
513  }
514  }
515  }
516  }
517  }
518  }
519  }
520  }
521  }
522  }
523  }
524  }
525  }
526  }
527  }
528  }
529  }
530  }
531  }
532  }
533  }
534  }
535  }
536  }
537  }
538  }
539  }
540  }
541  }
542  }
543  }
544  }
545  }
546  }
547  }
548  }
549  }
550  }
551  }
552  }
553  }
554  }
555  }
556  }
557  }
558  }
559  }
560  }
561  }
562  }
563  }
564  }
565  }
566  }
567  }
568  }
569  }
570  }
571  }
572  }
573  }
574  }
575  }
576  }
577  }
578  }
579  }
580  }
581  }
582  }
583  }
584  }
585  }
586  }
587  }
588  }
589  }
590  }
591  }
592  }
593  }
594  }
595  }
596  }
597  }
598  }
599  }
600  }
601  }
602  }
603  }
604  }
605  }
606  }
607  }
608  }
609  }
610  }
611  }
612  }
613  }
614  }
615  }
616  }
617  }
618  }
619  }
620  }
621  }
622  }
623  }
624  }
625  }
626  }
627  }
628  }
629  }
630  }
631  }
632  }
633  }
634  }
635  }
636  }
637  }
638  }
639  }
640  }
641  }
642  }
643  }
644  }
645  }
646  }
647  }
648  }
649  }
650  }
651  }
652  }
653  }
654  }
655  }
656  }
657  }
658  }
659  }
660  }
661  }
662  }
663  }
664  }
665  }
666  }
667  }
668  }
669  }
670  }
671  }
672  }
673  }
674  }
675  }
676  }
677  }
678  }
679  }
680  }
681  }
682  }
683  }
684  }
685  }
686  }
687  }
688  }
689  }
690  }
691  }
692  }
693  }
694  }
695  }
696  }
697  }
698  }
699  }
700  }
701  }
702  }
703  }
704  }
705  }
706  }
707  }
708  }
709  }
710  }
711  }
712  }
713  }
714  }
715  }
716  }
717  }
718  }
719  }
720  }
721  }
722  }
723  }
724  }
725  }
726  }
727  }
728  }
729  }
730  }
731  }
732  }
733  }
734  }
735  }
736  }
737  }
738  }
739  }
740  }
741  }
742  }
743  }
744  }
745  }
746  }
747  }
748  }
749  }
750  }
751  }
752  }
753  }
754  }
755  }
756  }
757  }
758  }
759  }
760  }
761  }
762  }
763  }
764  }
765  }
766  }
767  }
768  }
769  }
770  }
771  }
772  }
773  }
774  }
775  }
776  }
777  }
778  }
779  }
780  }
781  }
782  }
783  }
784  }
785  }
786  }
787  }
788  }
789  }
790  }
791  }
792  }
793  }
794  }
795  }
796  }
797  }
798  }
799  }
800  }
801  }
802  }
803  }
804  }
805  }
806  }
807  }
808  }
809  }
810  }
811  }
812  }
813  }
814  }
815  }
816  }
817  }
818  }
819  }
820  }
821  }
822  }
823  }
824  }
825  }
826  }
827  }
828  }
829  }
830  }
831  }
832  }
833  }
834  }
835  }
836  }
837  }
838  }
839  }
840  }
841  }
842  }
843  }
844  }
845  }
846  }
847  }
848  }
849  }
850  }
851  }
852  }
853  }
854  }
855  }
856  }
857  }
858  }
859  }
860  }
861  }
862  }
863  }
864  }
865  }
866  }
867  }
868  }
869  }
870  }
871  }
872  }
873  }
874  }
875  }
876  }
877  }
878  }
879  }
880  }
881  }
882  }
883  }
884  }
885  }
886  }
887  }
888  }
889  }
890  }
891  }
892  }
893  }
894  }
895  }
896  }
897  }
898  }
899  }
900  }
901  }
902  }
903  }
904  }
905  }
906  }
907  }
908  }
909  }
910  }
911  }
912  }
913  }
914  }
915  }
916  }
917  }
918  }
919  }
920  }
921  }
922  }
923  }
924  }
925  }
926  }
927  }
928  }
929  }
930  }
931  }
932  }
933  }
934  }
935  }
936  }
937  }
938  }
939  }
940  }
941  }
942  }
943  }
944  }
945  }
946  }
947  }
948  }
949  }
950  }
951  }
952  }
953  }
954  }
955  }
956  }
957  }
958  }
959  }
960  }
961  }
962  }
963  }
964  }
965  }
966  }
967  }
968  }
969  }
970  }
971  }
972  }
973  }
974  }
975  }
976  }
977  }
978  }
979  }
980  }
981  }
982  }
983  }
984  }
985  }
986  }
987  }
988  }
989  }
990  }
991  }
992  }
993  }
994  }
995  }
996  }
997  }
998  }
999  }
1000  }

```


PIDV	/ap/popmenus.c	Page 7
396	j += (spacing - 1) + 1;	
397	}	
398		
399		
400	k = menu->last_selection;	
401		
402	ok = -1;	
403	do {	
404		
405	if (k != ok) {	
406	if (ok != -1) {	
407	j = 1 + (ok * spacing);	
408		
409	Set_Text_Pos(j++, 0);	
410	Get_Char_Attr();	
411	Write_Char_Attr(character);	
412	while(!isalpha(character))	
413	{	
414	Set_Text_Pos(j++, 0);	
415	Get_Char_Attr();	
416	Write_Char_Attr(character);	
417	while(!isalpha(character))	
418	{	
419	Set_Text_Pos(j++, 0);	
420	Get_Char_Attr();	
421	Write_Char_Attr(character);	
422	while(!isalpha(character))	
423	{	
424	Set_Text_Pos(j++, 0);	
425	Get_Char_Attr();	
426	Write_Char_Attr(character);	
427	while(!isalpha(character))	
428	{	
429	Set_Text_Pos(j++, 0);	
430	Get_Char_Attr();	
431	Write_Char_Attr(character);	
432	while(!isalpha(character))	
433	{	
434	Set_Text_Pos(j++, 0);	
435	Get_Char_Attr();	
436	Write_Char_Attr(character);	
437	while(!isalpha(character))	
438	{	
439	Set_Text_Pos(j++, 0);	
440	Get_Char_Attr();	
441	Write_Char_Attr(character);	
442	while(!isalpha(character))	
443	{	
444	Set_Text_Pos(j++, 0);	
445	Get_Char_Attr();	
446	Write_Char_Attr(character);	
447	while(!isalpha(character))	
448	{	
449	Set_Text_Pos(j++, 0);	
450	Get_Char_Attr();	
451	Write_Char_Attr(character);	
452	while(!isalpha(character))	
453	{	
454	Set_Text_Pos(j++, 0);	
455	Get_Char_Attr();	
456	Write_Char_Attr(character);	
457	while(!isalpha(character))	
458	{	
459	Set_Text_Pos(j++, 0);	
460	Get_Char_Attr();	
461	Write_Char_Attr(character);	

PIDV	/ap/popmenus.c	Page 8
462	{	
463	capitalised = 0;	
464	while(
465	{ islower((int) menu->opt[j][capitalised]) }	
466	&& (menu->opt[j][capitalised] != '\0')	
467	capitalised++;	
468		
469	if (menu->opt[j][capitalised] == '\0')	
470	capitalised = 0;	
471	if (toupper(menu->opt[j][capitalised]) == toupper(key))	
472	{ option = j; break; }	
473		
474	if (option != -1) break;	
475		
476	key = Get_Next_Char();	
477	while(key != 0) {	
478		
479	if (!main_menu_flag) key = Get_Next_Char();	
480	else	
481	{ key = main_menu_key; main_menu_flag = 0; }	
482		
483	if (!escape && option == -1) {	
484		
485	if (!main_menu_flag) key = Get_Next_Char();	
486	else	
487	{ key = main_menu_key; main_menu_flag = 0; }	
488		
489		
490		
491	switch(key) {	
492		
493	case F1_HIT : wants_help = 1; break;	
494		
495	case F2_HIT : if (pop_fkey[0] != NULL)	
496	{	
497	last_opt_x = (maxx+1) - INPUT_LENGTH / 2;	
498	last_opt_y = ((maxy+1) / 2) - 1;	
499	if (!pop_fkey[0]) {	
500	break;	
501		
502	case F3_HIT : if (pop_fkey[1] != NULL)	
503	{	
504	last_opt_x = (maxx+1) - INPUT_LENGTH / 2;	
505	last_opt_y = ((maxy+1) / 2) - 1;	
506	if (!pop_fkey[1]) {	
507	break;	
508		
509	case F4_HIT : if (pop_fkey[2] != NULL)	
510	{	
511	last_opt_x = (maxx+1) - INPUT_LENGTH / 2;	
512	last_opt_y = ((maxy+1) / 2) - 1;	
513	if (!pop_fkey[2]) {	
514	break;	
515		
516	case F5_HIT : if (pop_fkey[3] != NULL)	
517	{	
518	last_opt_x = (maxx+1) - INPUT_LENGTH / 2;	
519	last_opt_y = ((maxy+1) / 2) - 1;	
520	if (!pop_fkey[3]) {	
521	break;	
522		
523		
524		
525		
526		
527		

```

528 case F6_HIT : if ( pop_fkey[4] != NULL )
529 {
530     last_opt_x = (maxx+1 - INPUT_LENGTH) / 2;
531     last_opt_y = ((maxy+1) / 2) - 1;
532     *((ff)pop_fkey[4])();
533     break;
534 }
535 case F7_HIT : if ( pop_fkey[5] != NULL )
536 {
537     last_opt_x = (maxx+1 - INPUT_LENGTH) / 2;
538     last_opt_y = ((maxy+1) / 2) - 1;
539     *((ff)pop_fkey[5])();
540     break;
541 }
542 case F8_HIT : if ( pop_fkey[6] != NULL )
543 {
544     last_opt_x = (maxx+1 - INPUT_LENGTH) / 2;
545     last_opt_y = ((maxy+1) / 2) - 1;
546     *((ff)pop_fkey[6])();
547     break;
548 }
549 case F9_HIT : if ( pop_fkey[7] != NULL )
550 {
551     last_opt_x = (maxx+1 - INPUT_LENGTH) / 2;
552     last_opt_y = ((maxy+1) / 2) - 1;
553     *((ff)pop_fkey[7])();
554     break;
555 }
556 case F10_HIT : if ( pop_fkey[8] != NULL )
557 {
558     last_opt_x = (maxx+1 - INPUT_LENGTH) / 2;
559     last_opt_y = ((maxy+1) / 2) - 1;
560     *((ff)pop_fkey[8])();
561     break;
562 }
563 case CURSORLEFT :
564 {
565     if ( k > 0 ) k--;
566     else k = number_of_options - 1;
567     break;
568 }
569 case CURSORRIGHT :
570 {
571     if ( k < ( number_of_options - 1 ) ) k++;
572     else k = 0;
573     break;
574 }
575 case CURSORDOWN :
576 {
577     option = k; break;
578 }
579 default :
580 {
581     break;
582 }
583 if ( !main_menu_key != 0 )
584 {
585     if ( !menu->tpl[ k ] == CALLS_MENU )
586     {
587         if ( k != ok ) {
588             if ( ok != -1 ) {
589                 j = 1 + ( ok * spacing );
590                 Set_Text_Pos( j++, 0 );
591                 Get_Char_Attr();
592             }
593         }
594     }

```

```

594 Write_Char_Attr( character,
595 LIGHTCYAN | ( BLUE << 4 ), 1 );
596 while( !isalpha( character ) )
597 {
598     Set_Text_Pos( j++, 0 );
599     Get_Char_Attr();
600     Write_Char_Attr( character,
601     YELLOW | ( BLUE << 4 ), 1 );
602 }
603 }
604 do
605 {
606     Set_Text_Pos( j++, 0 );
607     Get_Char_Attr();
608     Write_Char_Attr( character,
609     YELLOW | ( bar_colour << 4 ), 1 );
610     while( !isalpha( character ) );
611 } while( !escape );
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }

```

```

660 int i,j,k;
661 int o;
662 char c;
663 int length_of_title;
664 int right_bottom;
665 int maxx, maxy;
666 int video_mode;
667 int temp;
668 int number_of_options;
669 unsigned int menu_size_in_bytes;
670 byte* area_under_menu;
671 byte* copy_of_menu;
672 byte p;
673 int wants_help = 0;
674 int key;
675 int escape = 0;
676 int option = -1;
677 char* fl_is_help;
678 int capitalized;
679
680 fl_is_help = "-help";
681
682 if ( left < 0 ) left = 0;
683 if ( top < 0 ) top = 0;
684
685 video_mode = Get_Video_Mode();
686
687 maxx = Get_Text_X_Max( video_mode );
688 maxy = Get_Text_Y_Max( video_mode );
689
690 if ( maxx == -1 || maxy == -1 )
691 {
692     printf("Couldn't interpret video mode!\n");
693     exit(1);
694 }
695
696 Set_Text_Pos( 0, maxy );
697 Write_Char_Attr( 32, ( BLUE << 4 ), maxx );
698
699 i = 1;
700 Set_Text_Pos( j++, maxy );
701 Write_Char_Attr( 'P', LIGHTCYAN | ( BLUE << 4 ), 1 );
702 Set_Text_Pos( j++, maxy );
703 Write_Char_Attr( 'I', LIGHTCYAN | ( BLUE << 4 ), 1 );
704
705 i = 0;
706 while( (c = fl_is_help[i++]) != '\0' )
707 {
708     Set_Text_Pos( j++, maxy );
709     Write_Char_Attr( c, YELLOW | ( BLUE << 4 ), 1 );
710 }
711
712 for( k = 0; k < 8; k++ ) {
713     if ( pop_fkey[ k ] != NULL ) {
714         Set_Text_Pos( j++, maxy );
715         Write_Char_Attr( 32, LIGHTCYAN | ( BLUE << 4 ), 1 );
716         Set_Text_Pos( j++, maxy );
717         Write_Char_Attr( 'P', LIGHTCYAN | ( BLUE << 4 ), 1 );
718         Set_Text_Pos( j++, maxy );
719         Write_Char_Attr( 'I', LIGHTCYAN | ( BLUE << 4 ), 1 );
720         Set_Text_Pos( j++, maxy );
721         Write_Char_Attr( 'O', k+2, LIGHTCYAN | ( BLUE << 4 ), 1 );
722         Set_Text_Pos( j++, maxy );
723         Write_Char_Attr( '-', YELLOW | ( BLUE << 4 ), 1 );
724     }
725     i = 0;
726     while( (c = pop_fkey_name[k][i++]) != '\0' )
727     {
728         Set_Text_Pos( j++, maxy );
729         Write_Char_Attr( c, YELLOW | ( BLUE << 4 ), 1 );
730     }
731 }

```

```

726
727 if ( pop_fkey[ 8 ] != NULL ) {
728     Set_Text_Pos( j++, maxy );
729     Write_Char_Attr( 32, LIGHTCYAN | ( BLUE << 4 ), 1 );
730     Set_Text_Pos( j++, maxy );
731     Write_Char_Attr( 'P', LIGHTCYAN | ( BLUE << 4 ), 1 );
732     Set_Text_Pos( j++, maxy );
733     Write_Char_Attr( 'I', LIGHTCYAN | ( BLUE << 4 ), 1 );
734     Set_Text_Pos( j++, maxy );
735     Write_Char_Attr( 'O', LIGHTCYAN | ( BLUE << 4 ), 1 );
736     Set_Text_Pos( j++, maxy );
737     Write_Char_Attr( '-', YELLOW | ( BLUE << 4 ), 1 );
738 }
739
740 i = 0;
741 while( (c = pop_fkey_name[8][i++]) != '\0' )
742 {
743     Set_Text_Pos( j++, maxy );
744     Write_Char_Attr( c, YELLOW | ( BLUE << 4 ), 1 );
745 }
746
747 nx = 0; while( menu->title[nx] != '\0' ) nx++;
748 length_of_title = nx;
749
750 i = 0;
751
752 while ( strcmp( menu->opt[i], "-END" ) != 0 )
753 {
754     no = 0; while( menu->opt[i][no] != '\0' ) no++;
755     if ( no > nx ) nx = no;
756     i++;
757 }
758
759 number_of_options = 1;
760
761 ny = 1 + 1 /* For menu title */ + 2 /* For bar at top and bottom of menu */;
762
763 nx += 4 /* For bar at left and right of menu, and a space each side */;
764 right = left + nx - 1;
765 bottom = top + ny - 1;
766
767 menu_size_in_bytes = 2 /* one for data + one for attributes byte */ *
768     nx * ny;
769
770 area_under_menu = ( byte* ) Malloc( (unsigned long) menu_size_in_bytes );
771
772 if ( right > maxx )
773 {
774     temp = right - maxx;
775     right -= temp;
776     left -= temp;
777 }
778
779 if ( bottom > maxy ) {
780     temp = bottom - maxy;
781     bottom -= temp;
782     top -= temp;
783 }
784
785 p = area_under_menu;
786
787 for( i = top; i <= bottom; i++ )
788     for( j = left; j <= right; j++ )
789         Set_Text_Pos( j, i );
790
791

```



```

792 Get_Char_Attr();
793 *p++ = attribute;
794 *p++ = character;
795 }
796
797 Set_Text_Pos( left, top );
798 Write_Char_Attr( 218, WHITE, 1 );
799 Set_Text_Pos( left+1, top );
800 Set_Text_Pos( left+1, top );
801 Write_Char_Attr( 196, WHITE, (byte) (nx - 2) );
802 Write_Char_Attr( 191, WHITE, 1 );
803
804 Set_Text_Pos( left, top+1 );
805 Write_Char_Attr( 179, WHITE, 1 );
806
807 Set_Text_Pos( left+1, top+1 );
808 Write_Char_Attr( 32, 0, (byte) (nx - 2) );
809
810 } = left+(nx - length_of_title) / 2);
811 i = 0;
812 while( ( c = menu->title[i++] ) != '\0' )
813 {
814     Set_Text_Pos( j++, top+1 );
815     Write_Char_Attr( c, WHITE, 1 );
816 }
817
818 Set_Text_Pos( right, top+1 );
819 Write_Char_Attr( 179, WHITE, 1 );
820
821 for( k = 0; k < number_of_options; k++ )
822 {
823     Set_Text_Pos( left, top+2+k );
824     Write_Char_Attr( 179, WHITE, 1 );
825
826     j = left+1;
827
828     Set_Text_Pos( j++, top+2+k );
829     Write_Char_Attr( 32, YELLOW | ( BLUE << 4 ), 1 );
830
831     Set_Text_Pos( j++, top+2+k );
832     Write_Char_Attr( menu->oc[k][0], LIGHTCYAN | ( BLUE << 4 ), 1 );
833
834     i = 1;
835     while( ( c = menu->oc[k][i++] ) != '\0' )
836     {
837         Set_Text_Pos( j++, top+2+k );
838         Write_Char_Attr( c, YELLOW | ( BLUE << 4 ), 1 );
839     }
840
841     while( j < right )
842     {
843         Set_Text_Pos( j++, top+2+k );
844         Write_Char_Attr( 32, YELLOW | ( BLUE << 4 ), 1 );
845     }
846
847     Set_Text_Pos( right, top+2+k );
848     Write_Char_Attr( 179, WHITE, 1 );
849
850     }
851
852 Set_Text_Pos( left, bottom );
853 Write_Char_Attr( 192, WHITE, 1 );
854 Set_Text_Pos( left+1, bottom );
855 Write_Char_Attr( 196, WHITE, (byte) (nx - 2) );
856
857

```

```

858 Set_Text_Pos( right, bottom );
859 Write_Char_Attr( 217, WHITE, 1 );
860
861 k = menu->last_selection;
862
863 ok = -1;
864
865 do {
866     if ( k != ok ) {
867         if ( ok != -1 ) {
868             j = left+1;
869
870             Set_Text_Pos( j++, top+2+ok );
871             Get_Char_Attr();
872             Write_Char_Attr( character, ' ', 1 );
873             YELLOW | ( BLUE << 4 ), 1 );
874
875             Set_Text_Pos( j++, top+2+ok );
876             Get_Char_Attr();
877             Write_Char_Attr( character, ' ', 1 );
878             LIGHTCYAN | ( BLUE << 4 ), 1 );
879
880             while( j < right )
881             {
882                 Set_Text_Pos( j++, top+2+ok );
883                 Get_Char_Attr();
884                 Write_Char_Attr( character, ' ', 1 );
885                 YELLOW | ( BLUE << 4 ), 1 );
886             }
887
888             j = left+1;
889
890             while( j < right )
891             {
892                 Set_Text_Pos( j++, top+2+k );
893                 Get_Char_Attr();
894                 Write_Char_Attr( character, ' ', 1 );
895                 YELLOW | ( BLUE << 4 ), 1 );
896             }
897
898             ok = k;
899             Bye_Bye_Cursor();
900
901             if ( wants_help ) key = Help( menu, k, maxx, maxy );
902             else key = Get_Next_Char();
903
904             if ( key != 0 )
905             {
906                 if ( key == ENTER ) { option = k; break; }
907                 if ( key == ESCAPE ) { if ( wants_help )
908                     wants_help = 0;
909                     else { escape = 1;
910                         main_menu_Key = 0;
911                         break; }
912             }
913
914             For( j = 0; j < number_of_options; j++ )
915
916
917
918
919
920
921
922
923

```



```

1056 int keep_menu = 0;
1057
1058 current_menu = menu;
1059 current_option = option;
1060 menu->last_selection = option;
1061 main_menu_key = 0;
1062
1063 if ( menu->ctrl_option [ 4 ] ) { keep_menu = 1;
1064     last_opt_x = left+2;
1065     last_opt_y = top+2+option;
1066 }
1067
1068 else { last_opt_x = left; last_opt_y = top+2; }
1069
1070 if ( !keep_menu )
1071 {
1072     copy_of_menu = ( byte* ) Malloc( (unsigned long) menu_size_1
1073     n_bytes );
1074
1075     p = copy_of_menu;
1076
1077     for( i = top; i <= bottom; i++ )
1078     for( j = left; j <= right; j++ )
1079     {
1080         Set_Text_Pos( j, i );
1081         Get_Char_Attr();
1082         *p++ = attribute;
1083         *p++ = character;
1084     }
1085
1086     p = area_under_menu;
1087
1088     for( i = top; i <= bottom; i++ )
1089     for( j = left; j <= right; j++ )
1090     {
1091         Set_Text_Pos( j, i );
1092         attribute = *p++;
1093         character = *p++;
1094         Write_Char_Attr( character, attribute, i );
1095     }
1096
1097     Bye_Bye_Cursor();
1098
1099     if ( ( (menu->ctrl_option [ 4 ] & 3 ) == CALLS_FUNC )
1100     || ( (ctrl_menu->ctrl_option [ 1 ]) (option) ) )
1101     {
1102         if ( !keep_menu )
1103             Pop_Up_Menu( left+5,
1104             top+2+option,
1105             menu->ctrl_option [ 1 ] );
1106     }
1107     else
1108     {
1109         Pop_Up_Menu( left, top, menu->ctrl_option [ 1 ] );
1110         Bye_Bye_Cursor();
1111     }
1112
1113     if ( !keep_menu )
1114     {
1115         p = area_under_menu;
1116         for( i = top; i <= bottom; i++ )
1117         for( j = left; j <= right; j++ )
1118         {
1119             Set_Text_Pos( j, i );
1120

```

```

1121     Get_Char_Attr();
1122     *p++ = attribute;
1123     *p++ = character;
1124 }
1125
1126 p = copy_of_menu;
1127
1128 for( i = top; i <= bottom; i++ )
1129 for( j = left; j <= right; j++ )
1130 {
1131     Set_Text_Pos( j, i );
1132     attribute = *p++;
1133     character = *p++;
1134     Write_Char_Attr( character, attribute, i );
1135 }
1136
1137 Free( copy_of_menu );
1138
1139 Bye_Bye_Cursor();
1140
1141 }
1142
1143 option = -1;
1144
1145 while( !escape ) {
1146     p = area_under_menu;
1147
1148     for( i = top; i <= bottom; i++ )
1149     for( j = left; j <= right; j++ )
1150     {
1151         Set_Text_Pos( j, i );
1152         attribute = *p++;
1153         character = *p++;
1154         Write_Char_Attr( character, attribute, i );
1155     }
1156
1157     Free( area_under_menu );
1158
1159     int Help( menu, option, maxx, maxy )
1160     MENU* menu;
1161     int option;
1162     int maxx, maxy;
1163     {
1164         char* help;
1165         char* pi;
1166         int nx, ny;
1167         int nx;
1168         int cy;
1169         unsigned int help_size_in_bytes;
1170         byte* area_under_help;
1171         int i, j, k;
1172         int number_of_help_lines;
1173         int left, top, right, bottom;
1174         char* title;
1175         title = "help";
1176
1177         /* '*' marks new line, inserts a carriage return */
1178         help = p = menu->ctrl_option [ 1 ];
1179         nx = ny = 0;
1180

```

```

1187 do {
1188     nh = 0;
1189     while( ( c = *p++ ) != '\0' && c != 'i' ) nh++;
1190     ny++;
1191     if ( nh > nx ) nx = nh;
1192     while( c != '\0' ) {
1193         number_of_help_lines = ny;
1194         while( c != '\0' ) {
1195             nx += 2 /* for bar at left and right */
1196             + 2 /* for a space at left and right */
1197             ny += 1 /* for bar at top and bottom */
1198             + 1 /* for 'help' title */;
1199             if ( nx > ( maxx+1 ) ) { printf("Help text too wide for display mode\n");
1200                 exit(1); }
1201             if ( ny > ( maxy+1 ) ) { printf("Help text too long for display mode\n");
1202                 exit(1); }
1203             left = ( ( ( maxx+1 ) - nx ) / 2 ) - 1;
1204             top = ( ( ( maxy+1 ) - ny ) / 2 ) - 1;
1205             right = left + nx - 1;
1206             bottom = top + ny - 1;
1207             help_size_in_bytes = 2 /* one for data + one for attributes byte */ *
1208                 nx * ny;
1209             area_under_help = ( byte* ) Malloc( (unsigned long) help_size_in_bytes );
1210             p = area_under_help;
1211             for( i = top; i <= bottom; i++ )
1212                 for( j = left; j <= right; j++ )
1213                     Set_Text_Pos( j, i );
1214                     Get_Char_Attr( i );
1215                     *p++ = attribute;
1216                     *p++ = character;
1217             }
1218             Set_Text_Pos( left, top );
1219             Write_Char_Attr( 218, BLUE | ( LIGHTGRAY << 4 ), 1 );
1220             Set_Text_Pos( (left+1), top );
1221             Write_Char_Attr( 196, BLUE | ( LIGHTGRAY << 4 ), (byte) (nx - 2) );
1222             Set_Text_Pos( right, top );
1223             Write_Char_Attr( 191, BLUE | ( LIGHTGRAY << 4 ), 1 );
1224             Set_Text_Pos( left, top+1 );
1225             Write_Char_Attr( 179, BLUE | ( LIGHTGRAY << 4 ), 1 );
1226             Set_Text_Pos( left+1, top+1 );
1227             Write_Char_Attr( 32, BLUE | ( LIGHTGRAY << 4 ), (byte) (nx - 2) );
1228             j = left+(nx - 4) / 2;
1229             i = 0;
1230             while( ( c = title[i++] ) != '\0' )
1231                 {
1232

```

```

1253         Set_Text_Pos( j++, top+1 );
1254         Write_Char_Attr( c, WHITE | ( LIGHTGRAY << 4 ), 1 );
1255     }
1256     Set_Text_Pos( right, top+1 );
1257     Write_Char_Attr( 179, BLUE | ( LIGHTGRAY << 4 ), 1 );
1258     p = help;
1259     for( k = 0; k < number_of_help_lines; k++ )
1260     {
1261         Set_Text_Pos( left, top+2+k );
1262         Write_Char_Attr( 179, BLUE | ( LIGHTGRAY << 4 ), 1 );
1263         j = left+1;
1264         Set_Text_Pos( j++, top+2+k );
1265         Write_Char_Attr( 32, BLUE | ( LIGHTGRAY << 4 ), 1 );
1266         while( ( c = *p++ ) != '\0' && c != 'i' )
1267             {
1268                 Set_Text_Pos( j++, top+2+k );
1269                 Write_Char_Attr( 32, BLUE | ( LIGHTGRAY << 4 ), 1 );
1270                 while( ( c = *p++ ) != '\0' && c != 'i' )
1271                     {
1272                         Set_Text_Pos( j++, top+2+k );
1273                         Write_Char_Attr( c, BLUE | ( LIGHTGRAY << 4 ), 1 );
1274                     }
1275                 while( j < right )
1276                     {
1277                         Set_Text_Pos( j++, top+2+k );
1278                         Write_Char_Attr( 32, BLUE | ( LIGHTGRAY << 4 ), 1 );
1279                     }
1280                 Set_Text_Pos( right, top+2+k );
1281                 Write_Char_Attr( 179, BLUE | ( LIGHTGRAY << 4 ), 1 );
1282             }
1283             Set_Text_Pos( left, bottom );
1284             Write_Char_Attr( 192, BLUE | ( LIGHTGRAY << 4 ), 1 );
1285             Set_Text_Pos( (left+1), bottom );
1286             Write_Char_Attr( 196, BLUE | ( LIGHTGRAY << 4 ), (byte) (nx - 2) );
1287             Set_Text_Pos( right, bottom );
1288             Write_Char_Attr( 211, BLUE | ( LIGHTGRAY << 4 ), 1 );
1289             Bye_Bye_Cursor();
1290             c = Get_Next_Char();
1291             p = area_under_help;
1292             for( i = top; i <= bottom; i++ )
1293                 for( j = left; j <= right; j++ )
1294                     {
1295                         Set_Text_Pos( j, i );
1296                         attribute = *p++;
1297                         character = *p++;
1298                         Write_Char_Attr( character, attribute, 1 );
1299                     }
1300             Free( area_under_help );
1301             Bye_Bye_Cursor();
1302             return( c );
1303         }
1304         void Pop_Error( message )
1305

```

```

1319 char* message;
1320 char* pi;
1321 int nx,ny;
1322 int ci;
1323 int ci;
1324 unsigned int error_size_in_bytes;
1325 byte* area_under_error;
1326 int i,j,k;
1327 int left,top,right,bottom;
1328 char* title;
1329 int maxx,maxy;
1330 int video_mode;
1331 video_mode = Get_Video_Mode();
1332
1333 maxx = Get_Text_X_Max( video_mode );
1334 maxy = Get_Text_Y_Max( video_mode );
1335
1336 if ( maxx == -1 || maxy == -1 )
1337 {
1338     printf("Couldn't interpret video mode!\n");
1339     exit(1);
1340 }
1341
1342 title = "Error";
1343
1344 ny = 1;
1345
1346 nx = 0; while( message[ nx ] != '\0' ) nx++;
1347
1348 nx = 2 /* for bar at left and right */
1349 + 2 /* for a space at left and right */
1350
1351 ny = 2 /* for bar at top and bottom */
1352 + 1 /* for 'Error' title */
1353
1354 if ( nx > ( maxx+1 ) ) {
1355     message[ maxx - 6 ] = '\0';
1356     nx = 0; while( message[ nx ] != '\0' ) nx++;
1357     nx += 2 /* for bar at left and right */
1358     + 2 /* for a space at left and right */
1359 }
1360
1361 if ( ny > ( maxy+1 ) ) {
1362     Set_Text_Pos( 0,0 );
1363     printf("Error text too long for display mode!\n");
1364     exit(1);
1365 }
1366
1367 left = ( ( ( maxx+1 ) - nx ) / 2 ) - 1;
1368 top = ( ( ( maxy+1 ) - ny ) / 2 ) - 1;
1369 right = left + nx - 1;
1370 bottom = top + ny - 1;
1371
1372 error_size_in_bytes = 2 /* one for data + one for attributes byte */ *
1373     nx * ny;
1374
1375 area_under_error = ( byte* ) malloc( unsigned long( error_size_in_bytes ) );
1376
1377 P = area_under_error;
1378
1379

```

```

1385 for( i = top; i <= bottom; i++ )
1386     for( j = left; j <= right; j++ )
1387     {
1388         Set_Text_Pos( j, i );
1389         Get_Char_Attr( &ci );
1390         *p++ = attributes;
1391         *p++ = character;
1392     }
1393
1394 Set_Text_Pos( left, top );
1395 Write_Char_Attr( 218, WHITE | ( RED << 4 ), 1 );
1396 Set_Text_Pos( left+1, top );
1397 Write_Char_Attr( 196, WHITE | ( RED << 4 ), (byte) (nx - 2) );
1398 Set_Text_Pos( right, top );
1399 Write_Char_Attr( 191, WHITE | ( RED << 4 ), 1 );
1400
1401 Set_Text_Pos( left, top+1 );
1402 Write_Char_Attr( 179, WHITE | ( RED << 4 ), 1 );
1403
1404 Set_Text_Pos( left+1, top+1 );
1405 Write_Char_Attr( 32, WHITE | ( RED << 4 ), (byte) (nx - 2) );
1406
1407 j = left+(nx - 5) / 2;
1408 i = 0;
1409 while( ( ( c = title[i++] ) != '\0' ) && !isascii(c) )
1410 {
1411     Set_Text_Pos( j++, top+1 );
1412     Write_Char_Attr( c, WHITE | ( RED << 4 ), 1 );
1413 }
1414
1415 Set_Text_Pos( right, top+1 );
1416 Write_Char_Attr( 179, WHITE | ( RED << 4 ), 1 );
1417
1418 P = message;
1419
1420 Set_Text_Pos( left, top+2 );
1421 Write_Char_Attr( 179, WHITE | ( RED << 4 ), 1 );
1422
1423 j = left+1;
1424 Set_Text_Pos( j++, top+2 );
1425 Write_Char_Attr( 32, WHITE | ( RED << 4 ), 1 );
1426
1427 while( ( c = *p++ ) != '\0' && c != '\n' )
1428 {
1429     Set_Text_Pos( j++, top+2 );
1430     Write_Char_Attr( c, WHITE | ( RED << 4 ), 1 );
1431 }
1432
1433 while( j < right )
1434 {
1435     Set_Text_Pos( j++, top+2 );
1436     Write_Char_Attr( 32, WHITE | ( RED << 4 ), 1 );
1437 }
1438
1439 Set_Text_Pos( right, top+2 );
1440 Write_Char_Attr( 179, WHITE | ( RED << 4 ), 1 );
1441
1442 Set_Text_Pos( left, bottom );
1443 Write_Char_Attr( 196, WHITE | ( RED << 4 ), 1 );
1444 Set_Text_Pos( left+1, bottom );
1445 Write_Char_Attr( 196, WHITE | ( RED << 4 ), (byte) (nx - 2) );
1446 Set_Text_Pos( right, bottom );
1447 Write_Char_Attr( 191, WHITE | ( RED << 4 ), 1 );
1448
1449 P = area_under_error;
1450

```

```

1451     Bye_Bye_Cursor();
1452     while( (c = Get_Next_Char()) != ESCAPE )?
1453     {
1454         p = area_under_error;
1455         for( i = top; i <= bottom; i++)
1456             for( j = left; j <= right; j++)
1457             {
1458                 Set_Text_Pos( j, i );
1459                 attribute = *p++;
1460                 Write_Char_Attr( character, attribute, i );
1461             }
1462         Free( area_under_error );
1463     }
1464     Bye_Bye_Cursor();
1465 }
1466
1467 void Pop_Print( message, mode )
1468 char* message;
1469 char mode;
1470 {
1471     char* p;
1472     int nx,ny;
1473     int c;
1474     unsigned int print_size_in_bytes;
1475     byte* area_under_print;
1476     int i, j, k;
1477     int left, top, right, bottom;
1478     int max_x, max_y;
1479     int video_mode;
1480
1481     video_mode = Get_Video_Mode();
1482     maxx = Get_Text_X_Max( video_mode );
1483     maxy = Get_Text_Y_Max( video_mode );
1484     if ( maxx == -1 || maxy == -1 )
1485     {
1486         printf("Couldn't Interpret video mode!\n");
1487         exit(1);
1488     }
1489     nx = 0;
1490     while( message[ nx ] != '\0' ) nx++;
1491     ny = 1;
1492     while( message[ ny ] != '\0' ) ny++;
1493     nx += 2; /* for bar at left and right */
1494     ny += 2; /* for a space at left and right */
1495     ny += 2; /* for bar at top and bottom */
1496
1497     if ( nx > ( maxx+1 ) ) { printf("Error text too wide for display mode!\n");
1498                             exit(1); }
1499     if ( ny > ( maxy+1 ) ) { printf("Error text too long for display mode!\n");
1500                             exit(1); }
1501     left = ( ( maxx+1 ) - nx ) / 2 - 1;
1502     top = ( ( maxy+1 ) - ny ) / 2 - 1;
1503 }

```

```

1517     right = left + nx - 1;
1518     bottom = top + ny - 1;
1519     top_print = top;
1520     bottom_print = bottom;
1521     print_size_in_bytes = 2; /* one for data + one for attributes byte */
1522     if ( mode == 1 )
1523     {
1524         area_under_print = ( byte* ) Malloc( (unsigned long) print_size_in_bytes );
1525         p = area_under_print;
1526         for( i = top; i <= bottom; i++)
1527             for( j = left; j <= right; j++)
1528             {
1529                 Set_Text_Pos( j, i );
1530                 Get_Char_Attr( c );
1531                 *p++ = attribute;
1532                 *p++ = character;
1533             }
1534     }
1535     Set_Text_Pos( left, top );
1536     Write_Char_Attr( 218, WHITE | ( BLUE << 4 ), 1 );
1537     Set_Text_Pos( left+1, top );
1538     Write_Char_Attr( 196, WHITE | ( BLUE << 4 ), (byte) (nx - 2) );
1539     Set_Text_Pos( right, top );
1540     Write_Char_Attr( 191, WHITE | ( BLUE << 4 ), 1 );
1541     Set_Text_Pos( left, top+1 );
1542     Write_Char_Attr( 179, WHITE | ( BLUE << 4 ), 1 );
1543     Set_Text_Pos( left+1, top+1 );
1544     Write_Char_Attr( 32, WHITE | ( BLUE << 4 ), (byte) (nx - 2) );
1545     Set_Text_Pos( right, top+1 );
1546     Write_Char_Attr( 179, WHITE | ( BLUE << 4 ), 1 );
1547     p = message;
1548     Set_Text_Pos( left, top+1 );
1549     Write_Char_Attr( 179, WHITE | ( BLUE << 4 ), 1 );
1550     j = left+1;
1551     Set_Text_Pos( j++, top+1 );
1552     Write_Char_Attr( 32, WHITE | ( BLUE << 4 ), 1 );
1553     while( ( c = *p++ ) != '\0' && !isctrl(c) )
1554     {
1555         Set_Text_Pos( j++, top+1 );
1556         Write_Char_Attr( c, WHITE | ( BLUE << 4 ), 1 );
1557     }
1558     while( j < right )
1559     {
1560         Set_Text_Pos( j++, top+1 );
1561         Write_Char_Attr( c, WHITE | ( BLUE << 4 ), 1 );
1562     }
1563     Set_Text_Pos( j++, top+1 );
1564     Write_Char_Attr( 32, WHITE | ( BLUE << 4 ), 1 );
1565 }

```

PIDV	/ap/popmenus.c	Page 25
1583	Write_Char_Attr(179, WHITE (BLUE << 4), 1);	
1584		
1585	Set_Text_Pos(left, bottom);	
1586	Write_Char_Attr(192, WHITE (BLUE << 4), 1);	
1587	Set_Text_Pos(left+1, bottom);	
1588	Write_Char_Attr(196, WHITE (BLUE << 4), (byte) (mx - 2));	
1589	Set_Text_Pos(right, bottom);	
1590	Write_Char_Attr(217, WHITE (BLUE << 4), 1);	
1591		
1592	Bye_Bye_Cursor();	
1593		
1594	If (mode == 1) {	
1595		
1596	while((c = Get_Next_Char()) != ESCAPE);	
1597		
1598	p = area_under_print;	
1599		
1600	for(i = top; i <= bottom; i++)	
1601	{	
1602	for(j = left; j <= right; j++)	
1603	{	
1604	Set_Text_Pos(j, i);	
1605	attribute = *p++;	
1606	character = *p++;	
1607	Write_Char_Attr(character, attribute, 1);	
1608	}	
1609	Free(area_under_print);	
1610	}	
1611		
1612	Bye_Bye_Cursor();	
1613		
1614	char* Pop_Input(prompt)	
1615	{	
1616	char* prompt;	
1617		
1618	int maxx, maxy;	
1619	int video_mode;	
1620	int temp;	
1621	int top, left, right, bottom;	
1622	int prompt_length;	
1623	char* p;	
1624	byte* area_under_input;	
1625	unsigned int input_size_in_bytes;	
1626	int indent;	
1627	int not_escape = 1;	
1628	int x, y;	
1629	int i, j;	
1630	int c;	
1631		
1632	video_mode = Get_Video_Mode();	
1633	maxx = Get_Text_X_Max(video_mode);	
1634	maxy = Get_Text_Y_Max(video_mode);	
1635		
1636	If (maxx == -1 maxy == -1)	
1637	{	
1638	printf("Couldn't interpret video mode!\n");	
1639	exit(1);	
1640	}	
1641		
1642	top = last_opt_y;	
1643	left = last_opt_x;	
1644	right = left + INPUT_LENGTH - 1;	
1645	bottom = top + 2;	
1646		
1647		
1648	If (bottom > maxy) {	

PIDV	/ap/popmenus.c	Page 26
1649	temp = bottom - maxy;	
1650	top -= temp;	
1651	bottom -= temp;	
1652		
1653		
1654		
1655		
1656		
1657		
1658		
1659		
1660		
1661		
1662	input_size_in_bytes = 2 * (bottom - top + 1) * (right - left + 1);	
1663	area_under_input = (byte*) Malloc((unsigned long) input_size_in_bytes);	
1664		
1665	p = area_under_input;	
1666		
1667	for(i = top; i <= bottom; i++)	
1668	{	
1669	for(j = left; j <= right; j++)	
1670	{	
1671	Set_Text_Pos(j, i);	
1672	Get_Char_Attr();	
1673	*p++ = attribute;	
1674	}	
1675		
1676	i = 0; while(prompt[i] != '\0') i++;	
1677		
1678	prompt_length = i;	
1679		
1680	indent = (INPUT_LENGTH - 2) /* bar to left and right */ - prompt_length)/2;	
1681	j = left-1;	
1682	Set_Text_Pos(left, top);	
1683	Write_Char_Attr(216, YELLOW (BLUE << 4), 1);	
1684		
1685	for(i = 0; i < indent; i++)	
1686	{	
1687	Set_Text_Pos(j++, top);	
1688	Write_Char_Attr(196, YELLOW (BLUE << 4), 1);	
1689	}	
1690	for(i = 0; i < prompt_length; i++)	
1691	{	
1692	Set_Text_Pos(j++, top);	
1693	Write_Char_Attr(prompt[i], YELLOW (BLUE << 4), 1);	
1694	}	
1695		
1696	while(j < right)	
1697	{	
1698	Set_Text_Pos(j++, top);	
1699	Write_Char_Attr(196, YELLOW (BLUE << 4), 1);	
1700	}	
1701		
1702	Set_Text_Pos(right, top);	
1703	Write_Char_Attr(197, YELLOW (BLUE << 4), 1);	
1704	Set_Text_Pos(left, top+1);	
1705	Write_Char_Attr(179, YELLOW (BLUE << 4), 1);	
1706		
1707	Set_Text_Pos(left+1, top+1);	
1708	Write_Char_Attr(32, WHITE (BLUE << 4), 28);	
1709		
1710	Set_Text_Pos(right, top+1);	
1711	Write_Char_Attr(179, YELLOW (BLUE << 4), 1);	
1712		
1713	Set_Text_Pos(left, bottom);	
1714		

PIDV	/ap/popmenus.c	Page 27
1715	Write_Char_Attr(192, YELLOW (BLUE << 4), 1);	
1716	Set_Text_Pos((left+1), bottom);	
1717	Write_Char_Attr(196, YELLOW (BLUE << 4), 28);	
1718	Set_Text_Pos(right, bottom);	
1719	Write_Char_Attr(217, WHITE (BLUE << 4), 1);	
1720		
1721		
1722	x = left+1; y = top+1;	
1723	i = 0;	
1724	do	
1725	{	
1726	Set_Text_Pos(left+1+i, top+1);	
1727		
1728	while (!isalnum(c = Get_Next_Char()) &&	
1729	ispunct(c) &&	
1730	c != ENTER &&	
1731	c != DELETE &&	
1732	c != ESCAPE	
1733	c != ESCAPE	
1734);	
1735		
1736	if (c == ESCAPE) { not_escape = 0; break; }	
1737		
1738	if (c == ENTER)	
1739	{	
1740	pop_line[i] = '\0'; break;	
1741	}	
1742	else {	
1743	if (c == DELETE)	
1744	{	
1745	if (i > 0)	
1746	{ i--; x--;	
1747	Set_Text_Pos(x,y);	
1748	Write_Char_Attr(32, WHITE (BLUE << 4), 1);	
1749	}	
1750		
1751		
1752	else {	
1753	pop_line[i] = c;	
1754		
1755	Set_Text_Pos(x,y);	
1756	Write_Char_Attr(c, WHITE (BLUE << 4), 1);	
1757		
1758	x++; i++;	
1759	}	
1760		
1761		
1762	while(i < (INPUT_LENGTH - 3));	
1763		
1764	p = area_under_input;	
1765		
1766	for(i = top; i <= bottom; i++)	
1767	for(j = left; j <= right; j++)	
1768	{	
1769	Set_Text_Pos(j, i);	
1770	attribute = *p++;	
1771	character = *p++;	
1772	Write_Char_Attr(character, attribute, 1);	
1773		
1774	Free(area_under_input);	
1775		
1776	Bye_Bye_Cursor();	
1777		
1778	if (not_escape) return(pop_line);	
1779	else return('char');	
1780		

PIDV	/ap/popmenus.c	Page 28
1781		
1782	void Clear_Print()	
1783	{	
1784	int i;	
1785		
1786	for(i = top_print; i <= bottom_print; i++)	
1787	{	
1788	Set_Text_Pos(0, i);	
1789	Write_Char_Attr(32, 0, 80);	
1790	}	
1791		
1792		
1793		
1794	void Clear_Window()	
1795	{	
1796		
1797		
1798	void Print_Text_At(x,y, attrib, string)	
1799	int x,y;	
1800	int attrib;	
1801	char* string;	
1802	{	
1803		
1804	while (*string != '\0') {	
1805		
1806	Set_Text_Pos(x,y);	
1807	Write_Char_Attr(*string, attrib, 1);	
1808	x++;	
1809	string++;	
1810	}	
1811		
1812	void Print_Title(title)	
1813	char* title;	
1814	{	
1815	Print_Text_At(1,1, WHITE, title);	
1816		
1817		
1818		
1819		
1820	/*-----*/	
1821	Procedure : Print_Help	
1822		
1823	Arguments : string	
1824		
1825	Action :	
1826	Prints string on the text screen's help line.	
1827		
1828	/*-----*/	
1829		
1830	void Print_Help(string)	
1831	char* string;	
1832	{	
1833	Print_Text_At(1,23, BLUE (LIGHTGRAY << 4), string);	
1834		
1835	Bye_Bye_Cursor();	
1836		
1837		
1838	/*-----*/	
1839	Procedure : Clear_Help_Line	
1840		
1841	Arguments : None	
1842		
1843	Action :	
1844	Clears the text screen's help line.	
1845		
1846	/*-----*/	

```
1847 void Clear_Help_Line()
1848 {
1849     Print_Text_At( 1,23, 0,
1850     "
1851     );
1852     Bye_Bye_Cursor();
1853 }
1854
1855 /*-----*
1856 | Procedure : Print_Mouse_Usage
1857 |-----*
1858 | Arguments : None
1859 |-----*
1860 | Action :
1861 | Prints message on how to move the mouse and select menus.
1862 |-----*
1863 */
1864
1865 void Print_Mouse_Usage()
1866 {
1867     Print_Help(
1868     " Move select bar using cursor keys or mouse, select via enter or LEFT button "
1869     );
1870 }
1871
1872 /*-----*
1873 | Procedure : System_Command
1874 |-----*
1875 | Arguments : None
1876 |-----*
1877 | Action :
1878 | Permits execution of system commands from within the package.
1879 |-----*
1880 */
1881 void System_Command()
1882 {
1883     char* command;
1884     int x,y;
1885
1886     Print_Help (
1887     " You may now enter a MS-DOS system command e.g. dir "
1888     );
1889
1890     command = Pop_Input( "Enter command : " );
1891
1892     if ( command != NULL )
1893     {
1894         Set_Text_Page( 1 );
1895         System_Command();
1896         printf("\nPress ESCAPE to continue\n");
1897         while( Get_Next_Char() != ESCAPE );
1898     }
1899
1900     Set_Text_Page( 0 );
1901     Clear_Help_Line();
1902     Print_Mouse_Usage();
1903 }
1904
1905 }
1906
1907 }
1908 }
```



```
1 /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5 void Save_Screen();
6
```

```
1  /*
2  * Copyright (c) 1991 by Tom Judge.
3  * All rights reserved.
4  */
5  #include <stdio.h>
6  #include "savscrn.h"
7
8  void Save_Screen()
9  {
10     int x,y;
11     FILE *out;
12     char string[80];
13     char fname[80];
14     int first_time = 1;
15
16     gets( string );
17
18     sprintf( fname, "%s.raw", string );
19
20     out = fopen( fname, "wb" );
21
22     if ( out != NULL )
23     {
24         for( y = 0; y <= getmaxy(); y++ )
25             for( x = 0; x <= getmaxx(); x++ )
26             {
27                 if ( first_time == 1 )
28                    putc( getpixel( x, y ) | 0x80, out );
29                 else putc( getpixel( x, y ), out );
30             }
31         fclose( out );
32     }
33 }
34
35
```

```
1 #ifndef lint
2 static char sccsid[] = "@(#)tif_err.c 1.6 12/29/89";
3 #endif
4
5 /*
6  * Copyright (c) 1988 by Sam Leffler.
7  * All rights reserved.
8  *
9  * This file is provided for unrestricted use provided that this
10  * legend is included on all tape media and as a part of the
11  * software program in whole or part. Users may copy, modify or
12  * distribute this file at will.
13  */
14
15 /*
16  * TIFF Library.
17  */
18 #include <stdio.h>
19 #include "tiffio.h"
20 #include "popmenus.h"
21
22 #ifdef USE_PROTOTYPES
23 void TIFFError(char *module, char *fmt, ... )
24 #else
25 #define _VARARGS_
26 void TIFFError(module, fmt, va_alist)
27     char *module;
28     char *fmt;
29     va_dcl
30 #endif
31 {
32     va_list ap;
33     char err_string[78];
34     FILE *tmp;
35
36     VA_START(ap, fmt);
37     tmp = tmpfile();
38     vfprintf(tmp, fmt, ap);
39     va_end(ap);
40     rewind(tmp);
41     fgets(err_string, 78, tmp);
42     fclose(tmp);
43     err_string[77] = '\0';
44     Pop_Error(err_string);
45 }
46
47 }
```


PIDV	/api/tif_inf.c	Page 3
124	case PHOTOMETRIC MINISWHITE:	
125	Pop_Info_Line(" Photometric Interpretation: '\min-is-w'	
126	break;	
127	case PHOTOMETRIC MINISBLACK:	
128	Pop_Info_Line(" Photometric Interpretation: '\min-is-b'	
129	break;	
130	case PHOTOMETRIC RGB:	
131	Pop_Info_Line(" Photometric Interpretation: RGB color\	
132	n");	
133	case PHOTOMETRIC PALETTE:	
134	Pop_Info_Line(" Photometric Interpretation: palette co	
135	lor (RGB from colormap)\n");	
136	break;	
137	case PHOTOMETRIC MASK:	
138	Pop_Info_Line(" Photometric Interpretation: transparen	
139	cy mask\n");	
140	break;	
141	default:	
142	Pop_Info_Line(" Photometric Interpretation: %u (0x%x)\	
143	n";	
144	break;	
145	break;	
146	if (TIFFFieldSet(tif, FIELD_MATEING))	
147	Pop_Info_Line(" Mating: %s\n",	
148	td->td_mateing ? "alpha channel present" : "none");	
149	if (TIFFFieldSet(tif, FIELD_THRESHOLDING)) {	
150	switch (td->td_thresholding) {	
151	case THRESHOLD_BINARY:	
152	Pop_Info_Line(" Thresholding: binary art scan\n");	
153	break;	
154	case THRESHOLD_HALFTONE:	
155	Pop_Info_Line(" Thresholding: halftone or dithered sca	
156	n\n");	
157	break;	
158	case THRESHOLD_ERRORDIFFUSE:	
159	Pop_Info_Line(" Thresholding: error diffused\n");	
160	break;	
161	default:	
162	Pop_Info_Line(" Thresholding: %u (0x%x)\n",	
163	td->td_thresholding, td->td_thresholding);	
164	break;	
165	break;	
166	if (TIFFFieldSet(tif, FIELD_FILLORDER)) {	
167	switch (td->td_fillorder) {	
168	case FILLORDER_MSB2LSB:	
169	Pop_Info_Line(" Fillorder: msb-to-lsb\n");	
170	break;	
171	case FILLORDER_LSB2MSB:	
172	Pop_Info_Line(" Fillorder: lsb-to-msb\n");	
173	break;	
174	default:	
175	Pop_Info_Line(" Fillorder: %u (0x%x)\n",	
176	td->td_fillorder, td->td_fillorder);	
177	break;	
178	break;	
179	if (TIFFFieldSet(tif, FIELD_PREDICTOR))	
180	Pop_Info_Line(" Predictor: %u (0x%x)\n",	
181	td->td_predictor, td->td_predictor);	
182	if (TIFFFieldSet(tif, FIELD_ARTIST))	

PIDV	/api/tif_inf.c	Page 4
183	Pop_Info_Line(" Artist: %s\n", td->td_artist);	
184	if (TIFFFieldSet(tif, FIELD_DATETIME))	
185	Pop_Info_Line(" Date & Time: %s\n", td->td_datetime);	
186	if (TIFFFieldSet(tif, FIELD_HOSTCOMPUTER))	
187	Pop_Info_Line(" Host Computer: %s\n", td->td_hostcomputer);	
188	if (TIFFFieldSet(tif, FIELD_SOFTWARE))	
189	Pop_Info_Line(" Software: %s\n", td->td_software);	
190	if (TIFFFieldSet(tif, FIELD_DOCUMENTNAME))	
191	Pop_Info_Line(" Document Name: %s\n", td->td_documentname);	
192	if (TIFFFieldSet(tif, FIELD_IMAGEDESCRIPTION))	
193	Pop_Info_Line(" Image Description: %s\n",	
194	td->td_imagedescription);	
195	if (TIFFFieldSet(tif, FIELD_MAKE))	
196	Pop_Info_Line(" Make: %s\n", td->td_make);	
197	if (TIFFFieldSet(tif, FIELD_MODEL))	
198	Pop_Info_Line(" Model: %s\n", td->td_model);	
199	if (TIFFFieldSet(tif, FIELD_ORIENTATION)) {	
200	switch (td->td_orientation) {	
201	case ORIENTATION_TOPLEFT:	
202	Pop_Info_Line("row 0 top, col 0 lba\n");	
203	break;	
204	case ORIENTATION_TOPRIGHT:	
205	Pop_Info_Line("row 0 top, col 0 rba\n");	
206	break;	
207	case ORIENTATION_BOTTOMLEFT:	
208	Pop_Info_Line("row 0 bottom, col 0 lba\n");	
209	break;	
210	case ORIENTATION_BOTTOMRIGHT:	
211	Pop_Info_Line("row 0 bottom, col 0 rba\n");	
212	break;	
213	case ORIENTATION_LEFTTOP:	
214	Pop_Info_Line("row 0 lba, col 0 top\n");	
215	break;	
216	case ORIENTATION_LEFTBOT:	
217	Pop_Info_Line("row 0 lba, col 0 top\n");	
218	break;	
219	case ORIENTATION_RIGHTTOP:	
220	Pop_Info_Line("row 0 rba, col 0 top\n");	
221	break;	
222	case ORIENTATION_RIGHTBOT:	
223	Pop_Info_Line("row 0 rba, col 0 top\n");	
224	break;	
225	default:	
226	Pop_Info_Line("row 0 lba, col 0 bottom\n");	
227	break;	
228	td->td_orientation, td->td_orientation);	
229	break;	
230	break;	
231	if (TIFFFieldSet(tif, FIELD_SAMPLESPERPIXEL))	
232	Pop_Info_Line(" Samples per pixel: %u\n", td->td_samplesperpixel);	
233	if (TIFFFieldSet(tif, FIELD_BITSPERPIXEL))	
234	Pop_Info_Line(" Bits per pixel: %u\n", td->td_bitsperpixel);	
235	if (td->td_bitsperpixel < 0) {	
236	Pop_Info_Line(" Rows/strip: (infinite)\n");	
237	break;	
238	else	
239	Pop_Info_Line(" Rows/strip: %u\n", td->td_rowsperstrip	
240	if (TIFFFieldSet(tif, FIELD_MINSAMPLEVALUE))	
241	Pop_Info_Line(" Min Sample Value: %u\n", td->td_minsamplevalue	
242	if (TIFFFieldSet(tif, FIELD_MAXSAMPLEVALUE))	
243	Pop_Info_Line(" Max Sample Value: %u\n", td->td_maxsamplevalue	
244	if (TIFFFieldSet(tif, FIELD_PLANARCONFIG)) {	
245	switch (td->td_planarconfig) {	

PIDV	Jap/tif_int.c	Page 5
246	case PLANARCONFIG CONTIG:	
247	Pop_Info_Line(" Planar Configuration: single Image pia	
248	me\n");	
249	break;	
250	case PLANARCONFIG SEPARATE:	
251	Pop_Info_Line(" Planar Configuration: separate Image p	
252	ames\n");	
253	break;	
254	default:	
255	Pop_Info_Line(" Planar Configuration: %u (0x%x)\n",	
256	td->cd_planarconfig, td->cd_planarconfig);	
257	break;	
258	} else	
259	Pop_Info_Line(" Page Name: %s\n", td->cd_pagename);	
260	if (TIFFfieldset(tif, FIELD GRAYRESPONSEUNIT)) {	
261	Pop_Info_Line(" Gray Response Unit: %d\n",	
262	td->cd_grayresponseunit);	
263	else	
264	Pop_Info_Line(" %s\n",	
265	ResponseUnitNames[td->cd_grayresponseunit]);	
266	td->cd_grayresponseunit, td->cd_grayresponseunit);	
267	break;	
268	} else	
269	Pop_Info_Line(" %s\n",	
270	td->cd_grayresponseunit, td->cd_grayresponseunit);	
271	break;	
272	} else	
273	Pop_Info_Line(" Gray Response Curve: %d\n",	
274	td->cd_grayresponsecurve[1]);	
275	break;	
276	Pop_Info_Line(" %s\n",	
277	ResponseUnitNames[td->cd_grayresponseunit]);	
278	td->cd_grayresponseunit, td->cd_grayresponseunit);	
279	break;	
280	} else	
281	Pop_Info_Line(" %s\n",	
282	ResponseUnitNames[td->cd_grayresponseunit]);	
283	td->cd_grayresponseunit, td->cd_grayresponseunit);	
284	break;	
285	} else	
286	Pop_Info_Line(" Gray Response Curve: (present)\n");	
287	break;	
288	} else	
289	Pop_Info_Line(" Group 3 Options: %d\n",	
290	td->cd_group3options);	
291	break;	
292	} else	
293	Pop_Info_Line(" %s\n",	
294	ResponseUnitNames[td->cd_group3options]);	
295	td->cd_group3options, td->cd_group3options);	
296	break;	
297	} else	
298	Pop_Info_Line(" Group 4 Options: 0x%x\n",	
299	td->cd_group4options);	
300	break;	
301	} else	
302	Pop_Info_Line(" Resolution Unit: no meaningful units\n	
303	Resolution Unit: %d\n",	
304	td->cd_resolutionunit);	
305	break;	
306	} else	
307	Pop_Info_Line(" Resolution Unit: centimeter\n");	

PIDV	Jap/tif_int.c	Page 6
308	break;	
309	Pop_Info_Line(" Resolution Unit: %u (0x%x)\n",	
310	td->cd_resolutionunit, td->cd_resolutionunit);	
311	break;	
312	} else	
313	Pop_Info_Line(" %s\n",	
314	ResponseUnitNames[td->cd_colorresponseunit]);	
315	td->cd_colorresponseunit, td->cd_colorresponseunit);	
316	break;	
317	} else	
318	Pop_Info_Line(" Page Number: %u\n",	
319	td->cd_pagenumber);	
320	break;	
321	} else	
322	Pop_Info_Line(" Color Response Unit: %d\n",	
323	td->cd_colorresponseunit);	
324	break;	
325	} else	
326	Pop_Info_Line(" %s\n",	
327	ResponseUnitNames[td->cd_colorresponseunit]);	
328	td->cd_colorresponseunit, td->cd_colorresponseunit);	
329	break;	
330	} else	
331	Pop_Info_Line(" %u (0x%x)\n",	
332	td->cd_colorresponseunit, td->cd_colorresponseunit);	
333	break;	
334	} else	
335	Pop_Info_Line(" Color Map: %d\n",	
336	td->cd_colormap);	
337	break;	
338	} else	
339	Pop_Info_Line(" %s\n",	
340	ResponseUnitNames[td->cd_colormap]);	
341	td->cd_colormap, td->cd_colormap);	
342	break;	
343	} else	
344	Pop_Info_Line(" (present)\n");	
345	break;	
346	} else	
347	Pop_Info_Line(" Color Response Curve: %d\n",	
348	td->cd_colorresponsecurve[1]);	
349	break;	
350	} else	
351	Pop_Info_Line(" %s\n",	
352	ResponseUnitNames[td->cd_colorresponsecurve]);	
353	td->cd_colorresponsecurve, td->cd_colorresponsecurve);	
354	break;	
355	} else	
356	Pop_Info_Line(" (present)\n");	
357	break;	
358	} else	
359	Pop_Info_Line(" %s\n",	
360	ResponseUnitNames[td->cd_colorresponsecurve]);	
361	td->cd_colorresponsecurve, td->cd_colorresponsecurve);	
362	break;	

```
1  #ifndef lint
2  static char sccsid[] = "@(#)tif_error.c 1.6 12/29/89"
3  #endif
4
5  /*
6  * Copyright (c) 1988 by Sam Leffler.
7  * All rights reserved.
8  *
9  * This file is provided for unrestricted use provided that this
10 * legend is included on all tape media and as a part of the
11 * software program in whole or part. Users may copy, modify or
12 * distribute this file at will.
13 */
14
15 /*
16 * TIFF Library.
17 */
18 #include <stdio.h>
19 #include "tiffio.h"
20 #include "popmenus.h"
21
22 #ifdef USE_PROTOTYPES
23 void TIFFWarning(char *module, char *fmt, ... )
24 #else
25 /*VARARGS2*/
26 void TIFFWarning(module, fmt, va_alist)
27     char *module;
28     char *fmt;
29     va_dcl
30 #endif
31 {
32     va_list ap;
33     char err_string[78];
34     FILE *tmp;
35
36     VA_START(ap, fmt);
37     tmp = tmpfile();
38     vfprintf(tmp, fmt, ap);
39     va_end(ap);
40     fseek(tmp, 0, 2);
41     fprintf(err_string, 78, tmp);
42     fclose(tmp);
43     err_string[77] = '\0';
44     Pop_Error(err_string, 1);
45 }
46
47 }
```

THE BRITISH LIBRARY

BRITISH THESIS SERVICE

VOL 2

TITLE QUANTITATIVE DIGITAL IMAGE PROCESSING IN
FRINGE ANALYSIS AND PARTICLE IMAGE
VELOCIMETRY (PIV)

AUTHOR Thomas Richard
JUDGE

DEGREE Ph.D

AWARDING Warwick University
BODY

DATE 1992

THESIS DX177940
NUMBER

THIS THESIS HAS BEEN MICROFILMED EXACTLY AS RECEIVED

The quality of this reproduction is dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction. Some pages may have indistinct print, especially if the original papers were poorly produced or if awarding body sent an inferior copy. If pages are missing, please contact the awarding body which granted the degree.

Previously copyrighted materials (journals articles, published texts etc.) are not filmed.

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no information derived from it may be published without the author's prior written consent.

Reproduction of this thesis, other than as permitted under the United Kingdom Copyright Designs and Patents Act 1988, or under specific agreement with the copyright holder, is prohibited.

C 7

DX VOL. 2

177940