The Application Of Artificial Neural Networks And Genetic Algorithms To The Estimation Of Electrode Response Characteristics And Stability Constants

Volume 2

By Margaret Kathleen Hartnett, School of Chemical Sciences, Dublin City University

Supervised by Dr. Dermot Diamond, School of Chemical Sciences, Dublin City University

A thesis submitted for the degree of Doctor of Philosophy

August 1994

Volume 2 APPENDICES

FIGURES

Appendix 1 Simple FIA patterns used for restricted training in chapter 2 1

Appendix 2 Patterns used for studies of noise addition in chapter 2 3

Appendix 3 Test patterns used for the study of the effect of peak height variation on pattern classification in chapter 2 20

Appendix 4 Test patterns used to study the influence of baseline shifting on pattern classification in chapter 2 29

Appendix 5 Distorted patterns used for broad training in chapter 2 33

Appendix 6 Patterns used for testing the network described in chapter 2 57

Appendix 7 TABLES

Table 2.1 Classification of patterns in appendix 2 by a network with 55 neurons in its hidden layer trained with the simple FIA patterns 76

Table 2.2 Classification results from the network described for table 2.1 to a test set produced by reducing the heights of valid FIA peaks............. 77

Table 2.3 Classification of patterns in appendix 2 by a network with 55 neurons in its hidden layer trained with the distorted FIA patterns...................................78

Table 2.4 Classification results from the network described for table 2.3 to a test set produced by reducing the heights of valid FIA peaks............. 79

Table 3.1 Activities of ammonium, sodium, potassium and calcium in the calibration solutions in chapter 3 (corresponding to the concentrations seen in table 3.1 of chapter 3)... 80

Appendix 8 Rank Scaling For GA In Chapter 3 81

APPENDIX 9 SOFTWARE

RANDHEAD.C 82

GENCA5.C 86

SIMPLEX.C 116

INVERT.C 127

GASTAB.C* 135

GASIMP.C* 169

STDEV.C* 191

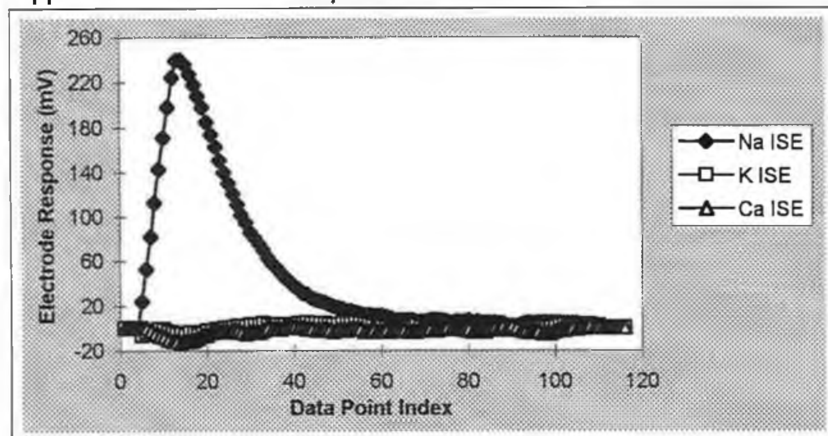
Appendix 10 Formal Framework For Genetic Algorithms 217

Acknowledgements

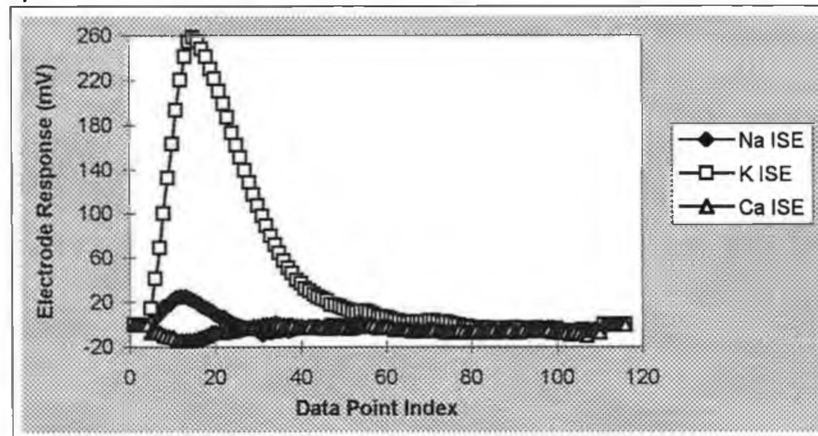
---

* The use of the program EQUIL whose source code is included in programs GASTAB.C, GASIMP.C and STDEV.C is gratefully acknowledged. For further information of this program contact M. Bos at the Laboratory For Chemical Analysis in the Chemical Technology department of the University of Twente.

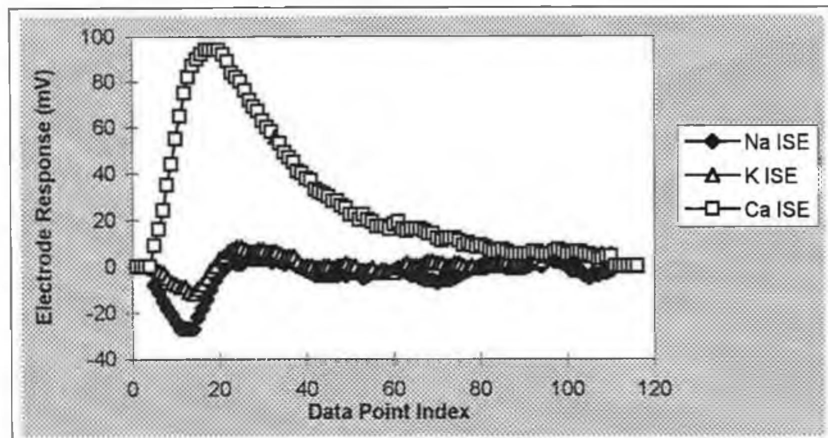
Appendix 1 Simple FIA Patterns Used For Restricted Training In Chapter 2



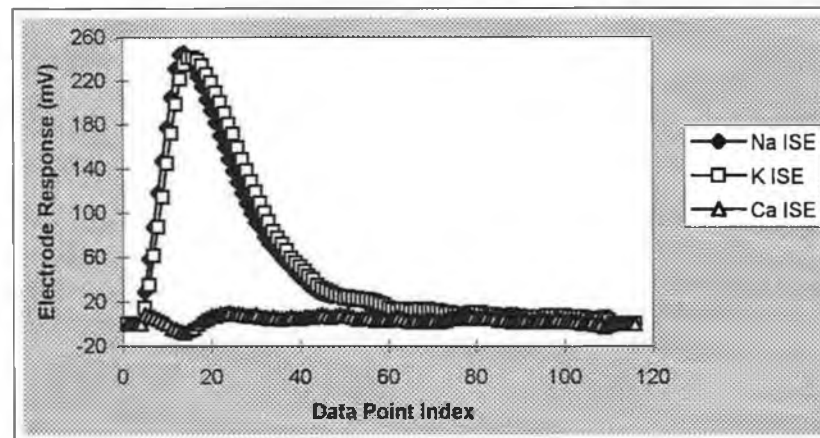
Pattern 1.1
FIA trace to a solution containing 0.1 M sodium (key = Na in table 2.1)



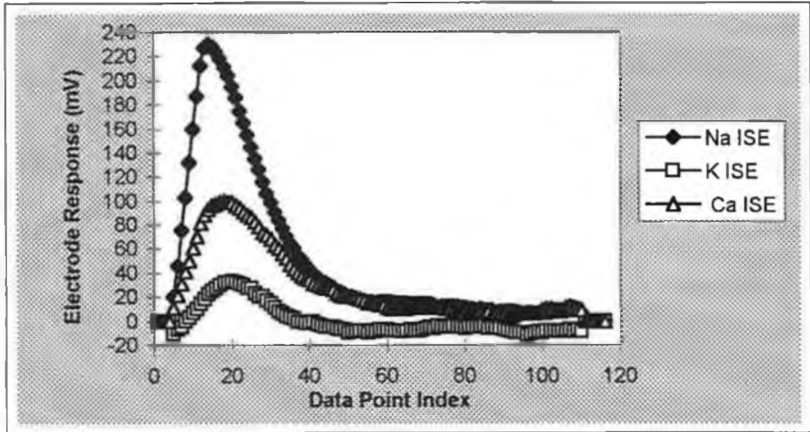
Pattern 1.2
FIA trace to a solution containing 0.1 M potassium (key = K in table 2.1)



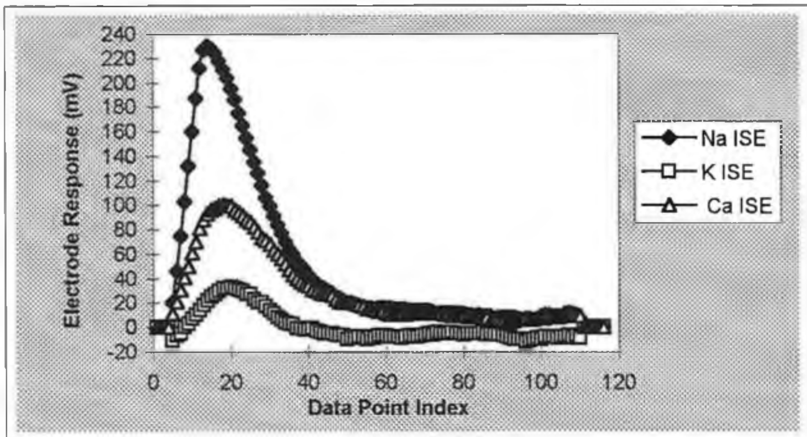
Pattern 1.3
FIA trace to a solution containing 0.1 M calcium (key = Ca in table 2.1)



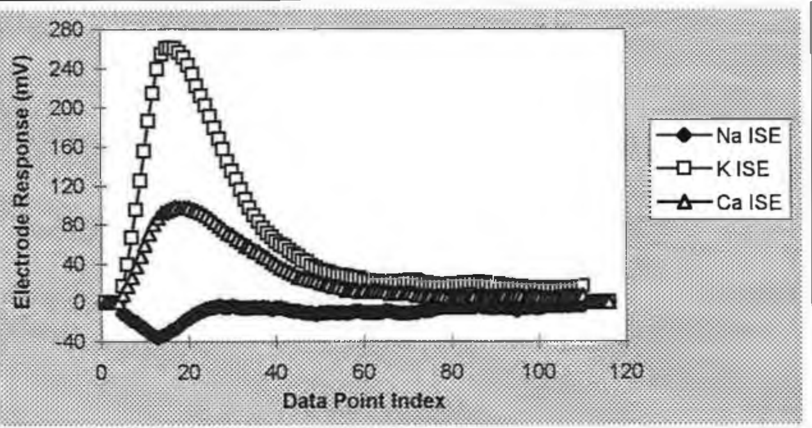
Pattern 1.4
FIA trace to a solution containing 0.1 M sodium and potassium (key = NaK in table 2.1)



Pattern 1.5
FIA trace to a solution containing 0.1 M sodium and calcium (key = NaCa in table 2.1)



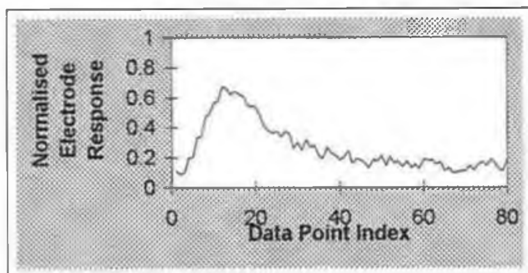
Pattern 1.7
FIA trace to a solution containing 0.1 M sodium, potassium and calcium (key = NaKCa in table 2.1)



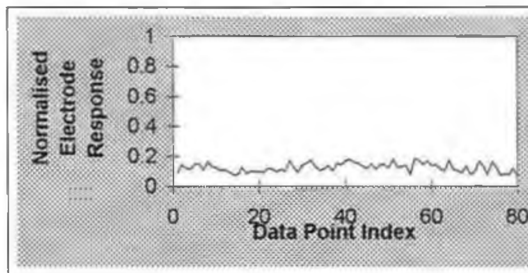
Pattern 1.6
FIA trace to a solution containing 0.1 M potassium and calcium
(key = KCa in table 2.1)

Appendix 2 Patterns Used For Studies Of Noise Addition In Chapter 2 [See 2.3.1.3 and 2.3.2.2]

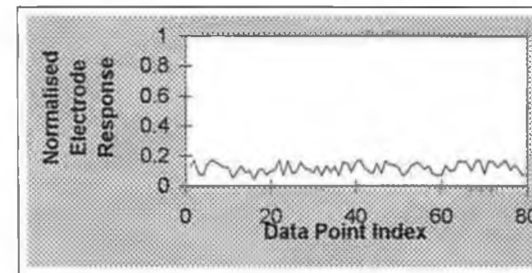
Pattern 1



Na ISE

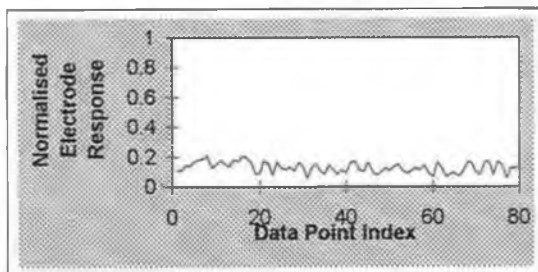


K ISE

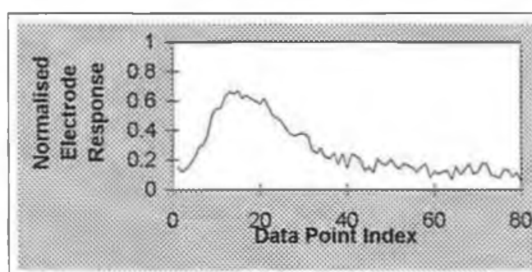


Ca ISE

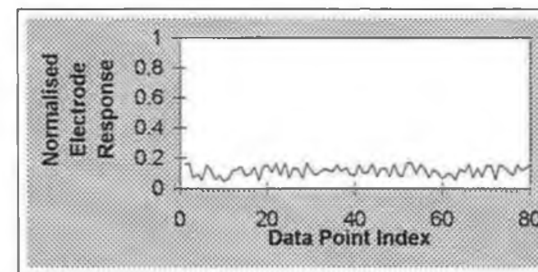
Pattern 2



Na ISE

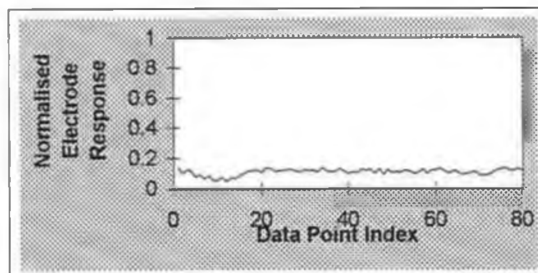


K ISE

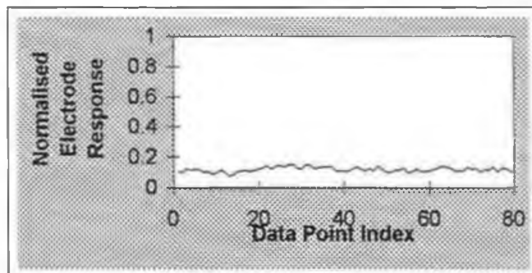


Ca ISE

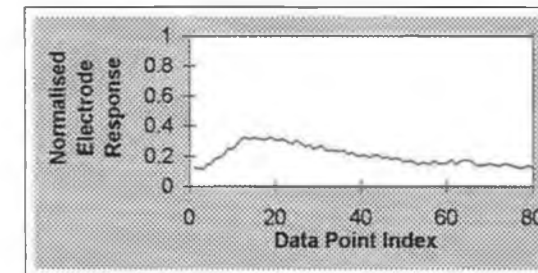
Pattern 3



Na ISE

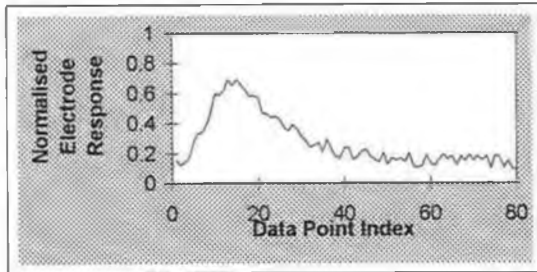


K ISE

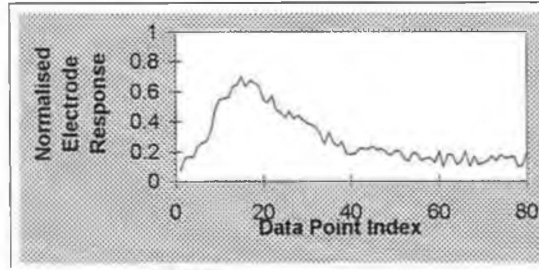


Ca ISE

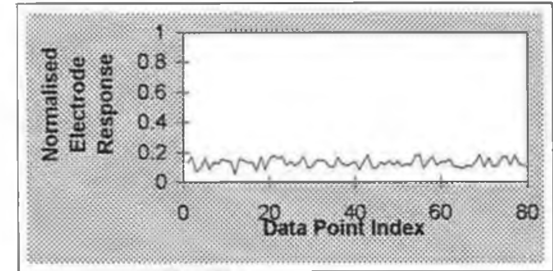
Pattern 4



Na ISE

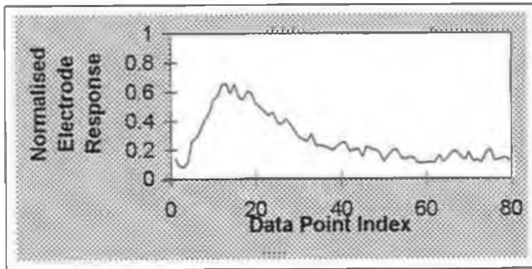


K ISE

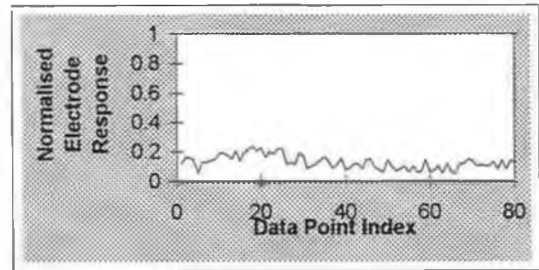


Ca ISE

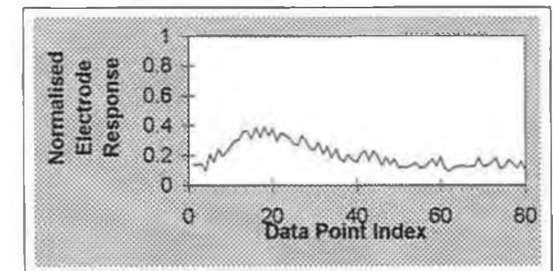
Pattern 5



Na ISE

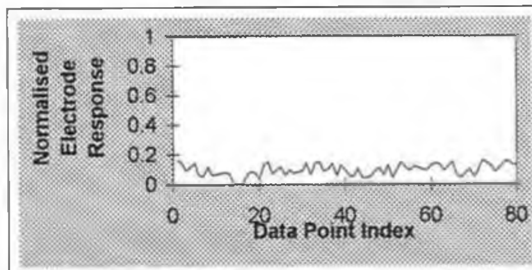


K ISE

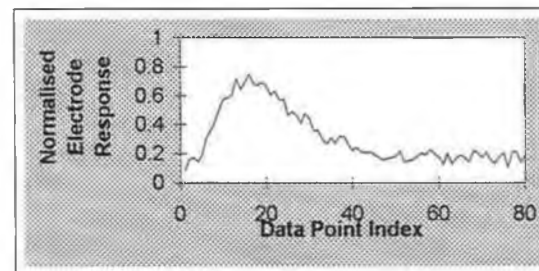


Ca ISE

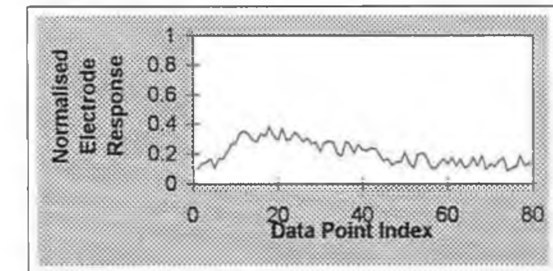
Pattern 6



Na ISE

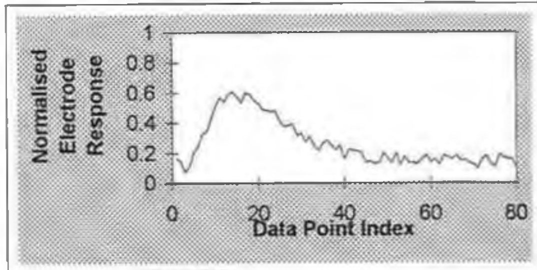


K ISE

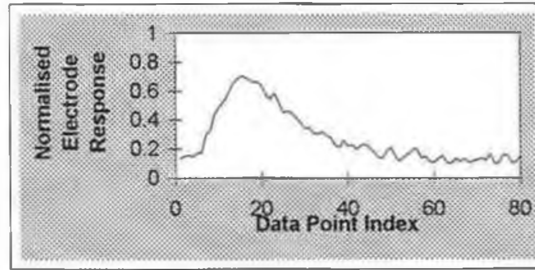


Ca ISE

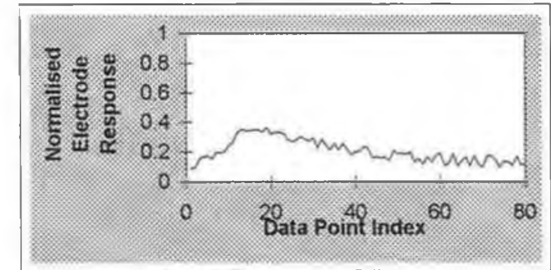
Pattern 7



Na ISE

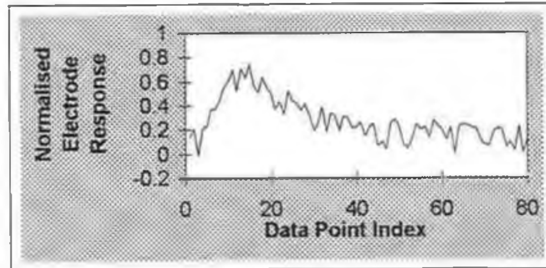


K ISE

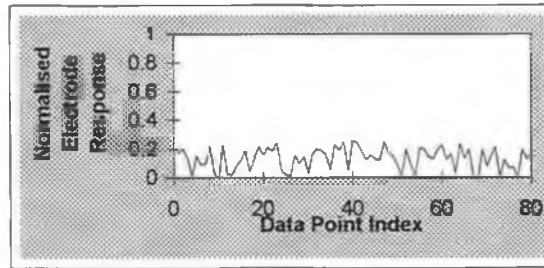


Ca ISE

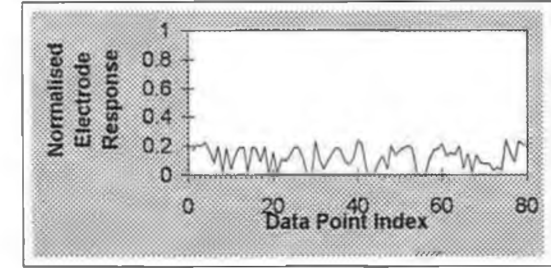
Pattern 8



Na ISE

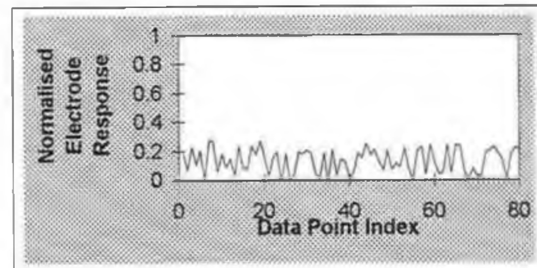


K ISE

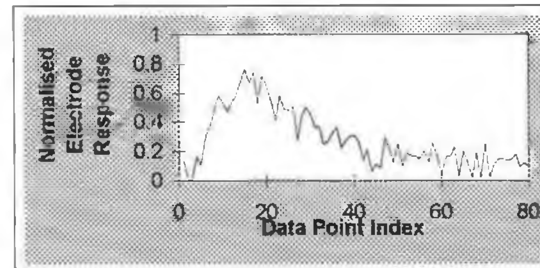


Ca ISE

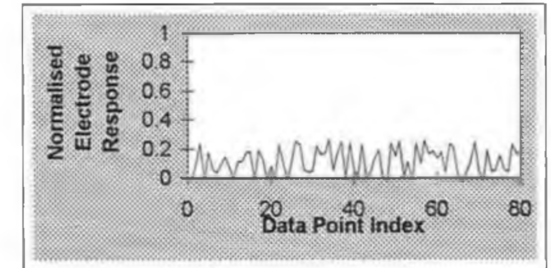
Pattern 9



Na ISE

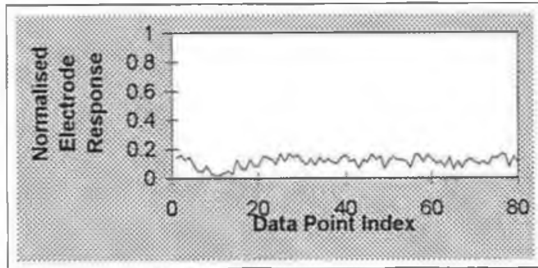


K ISE

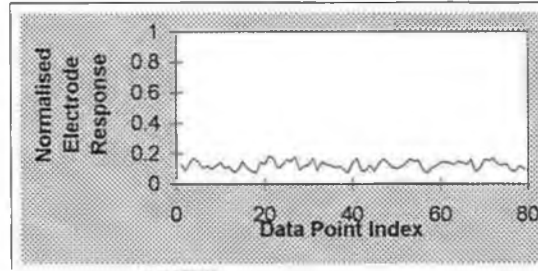


Ca ISE

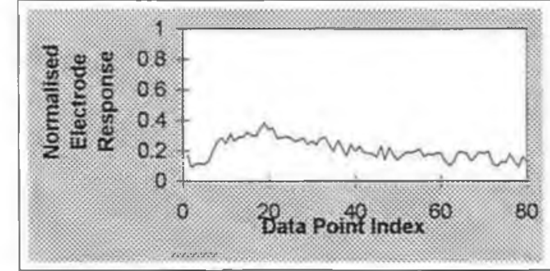
Pattern 10



Na ISE

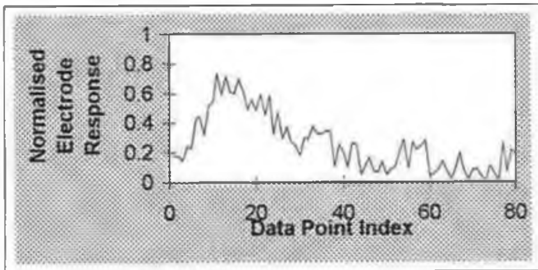


K ISE

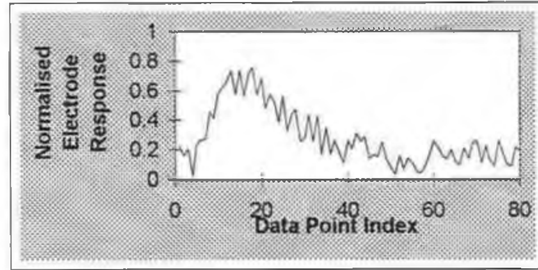


Ca ISE

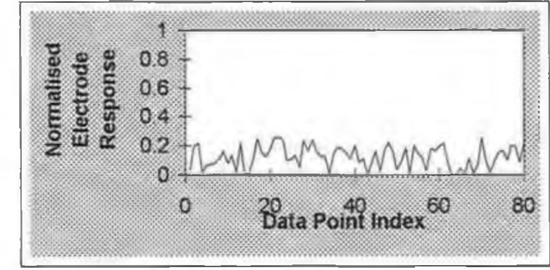
Pattern 11



Na ISE

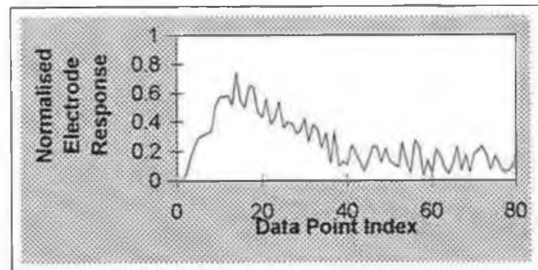


K ISE

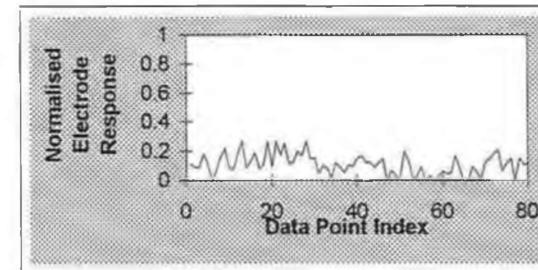


Ca ISE

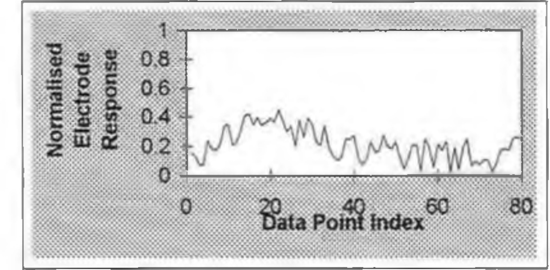
Pattern 12



Na ISE

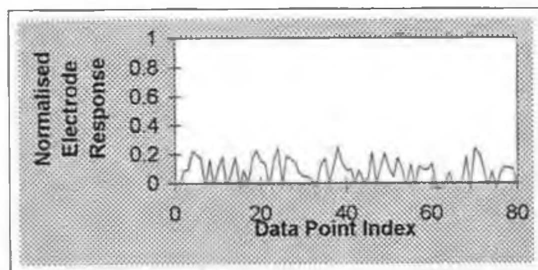


K ISE

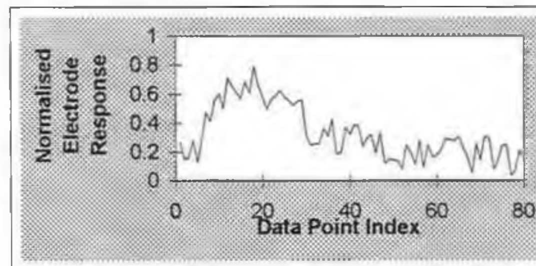


Ca ISE

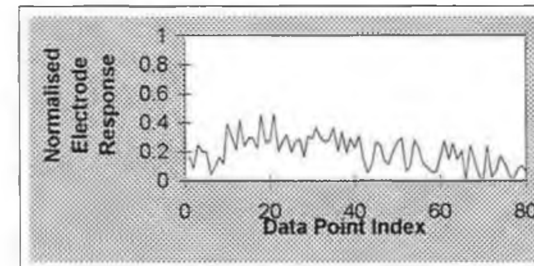
Pattern 13



Na ISE

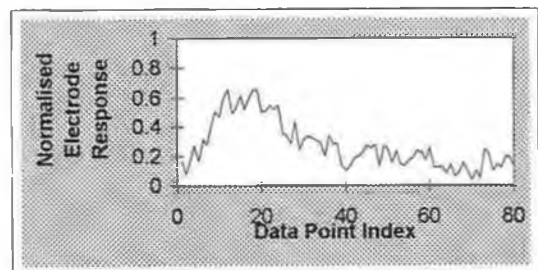


K ISE

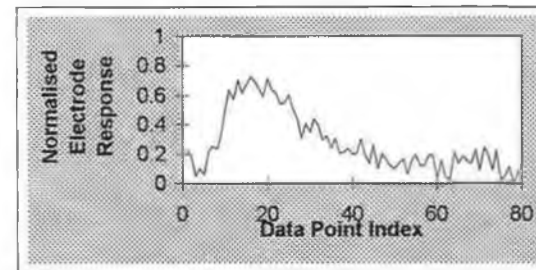


Ca ISE

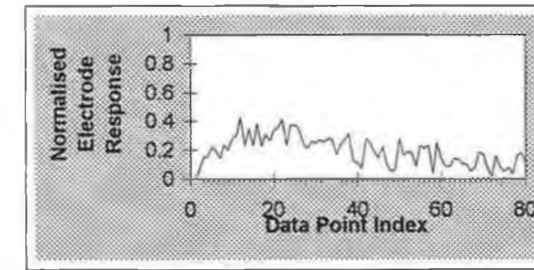
Pattern 14



Na ISE

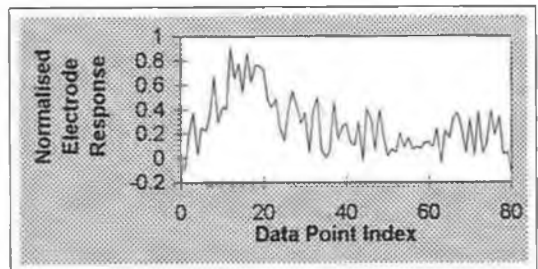


K ISE

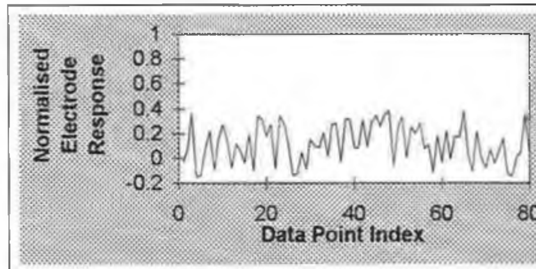


Ca ISE

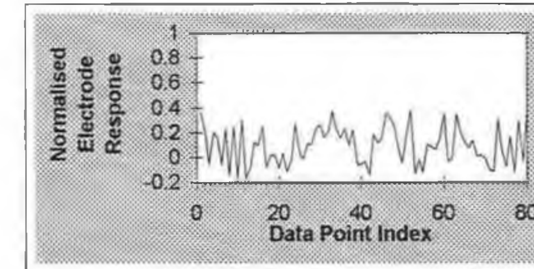
Pattern 15



Na ISE

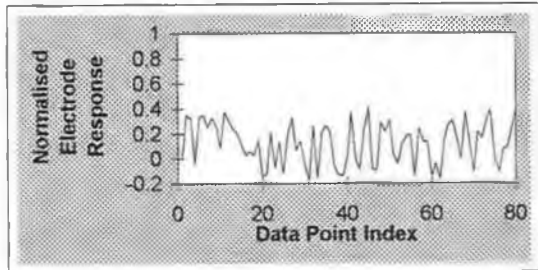


K ISE

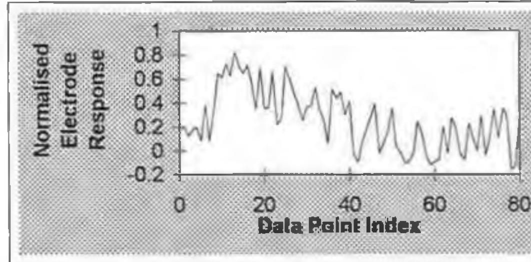


Ca ISE

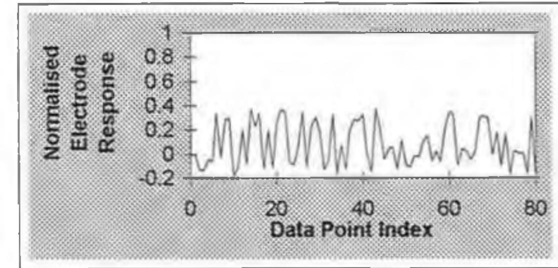
Pattern 16



Na ISE

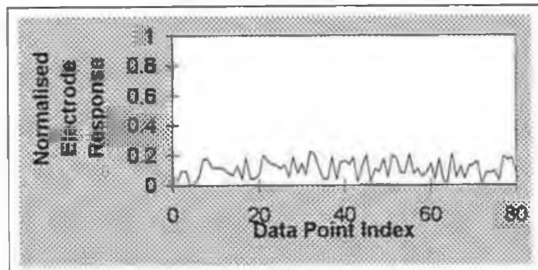


K ISE

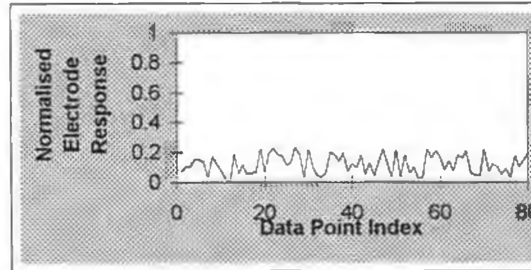


Ca ISE

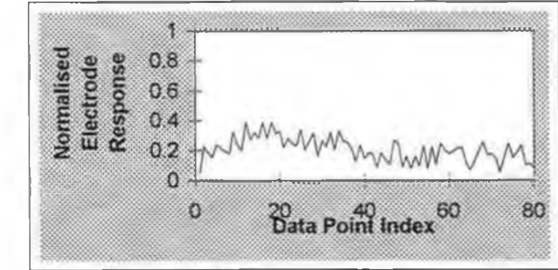
Pattern 17



Na ISE

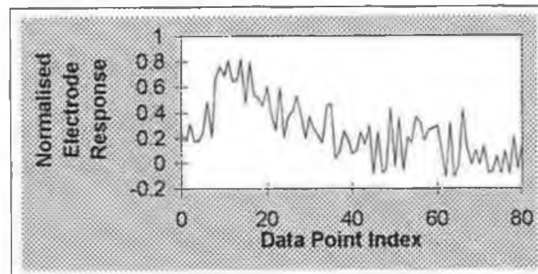


K ISE

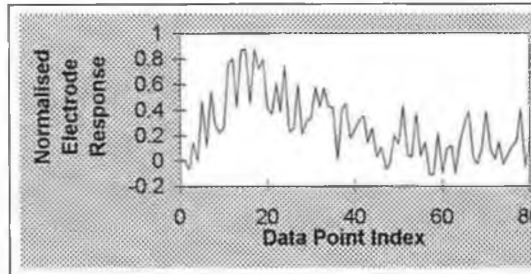


Ca ISE

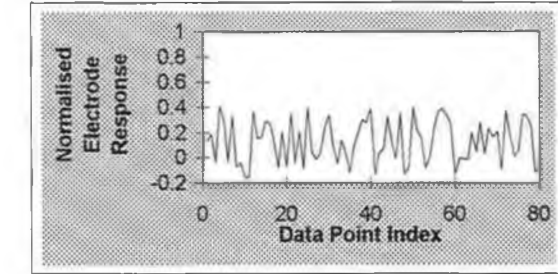
Pattern 18



Na ISE

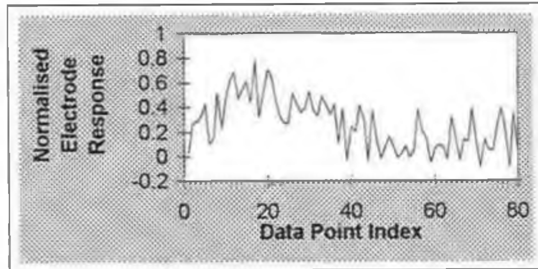


K ISE

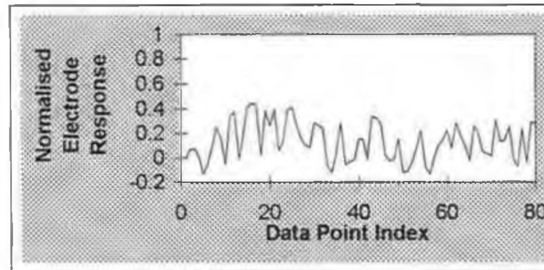


Ca ISE

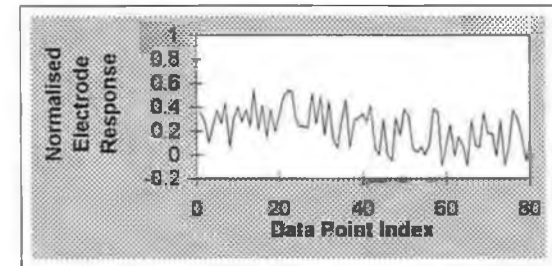
Pattern 19



Na ISE

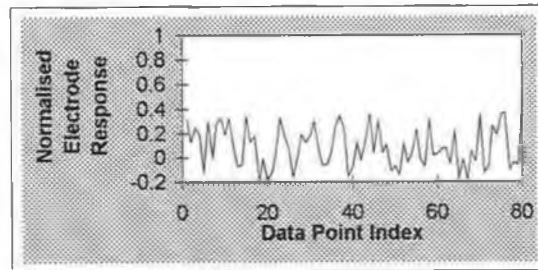


K ISE

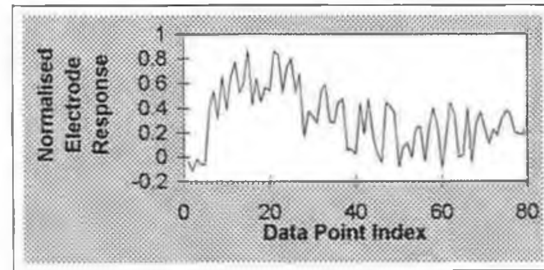


Ca ISE

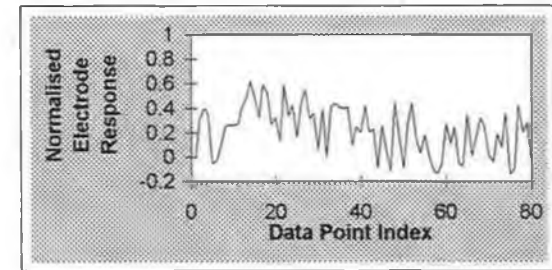
Pattern 20



Na ISE

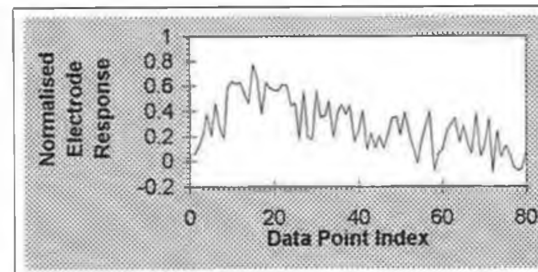


K ISE

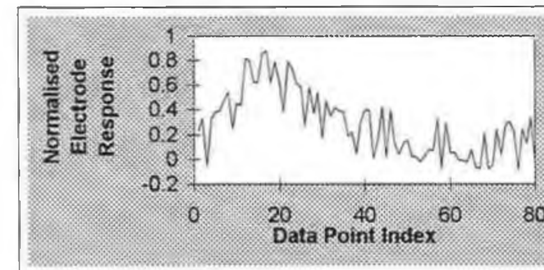


Ca ISE

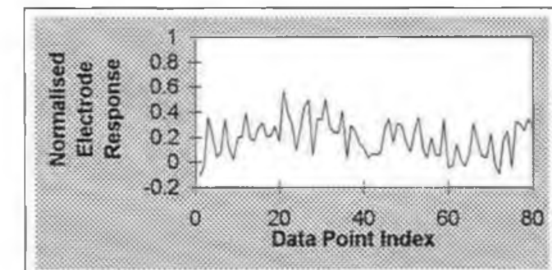
Pattern 21



Na ISE

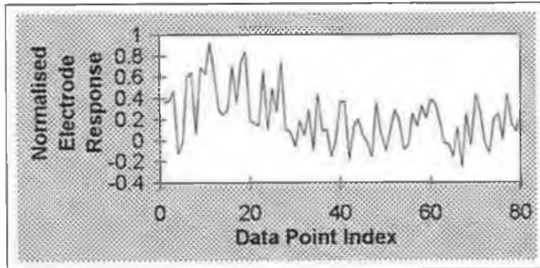


K ISE

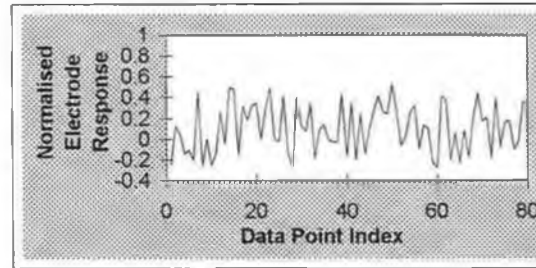


Ca ISE

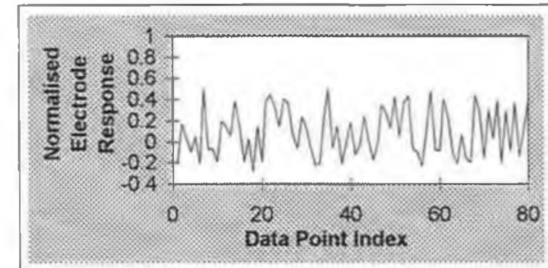
Pattern 22



Na ISE

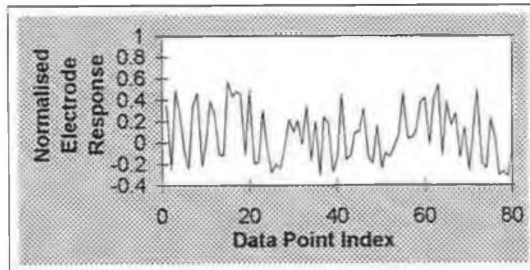


K ISE

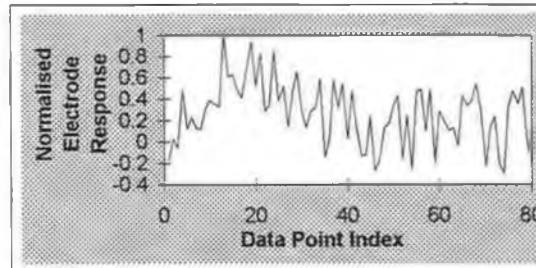


Ca ISE

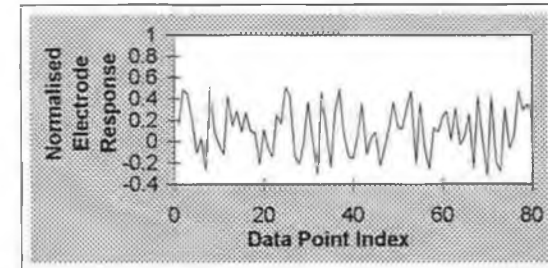
Pattern 23



Na ISE

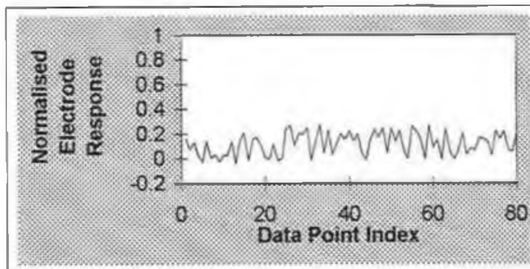


K ISE

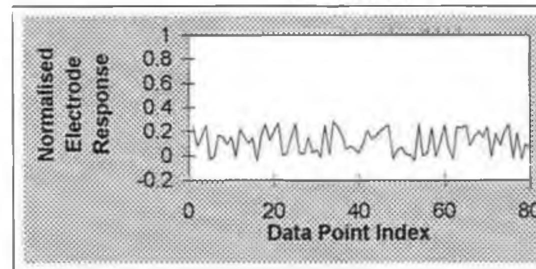


Ca ISE

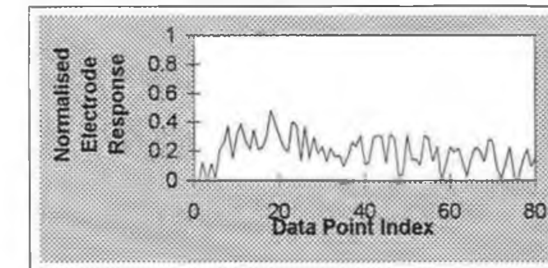
Pattern 24



Na ISE

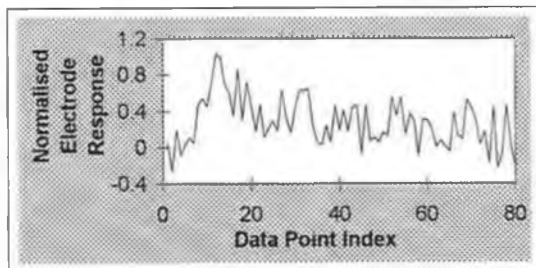


K ISE

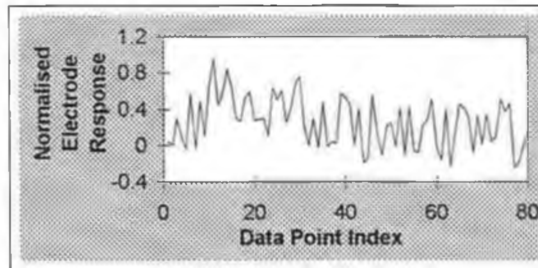


Ca ISE

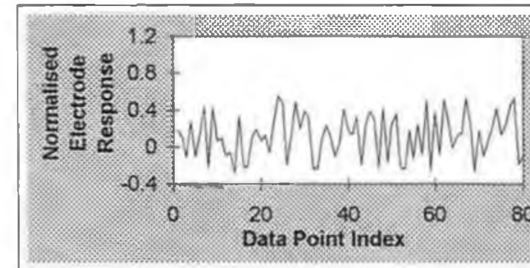
Pattern 25



Na ISE

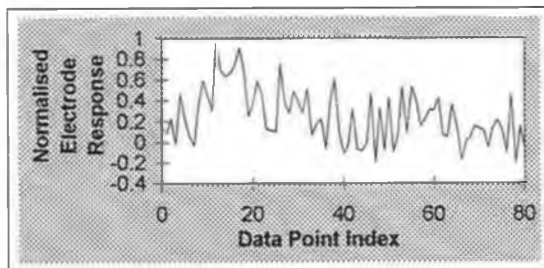


K ISE

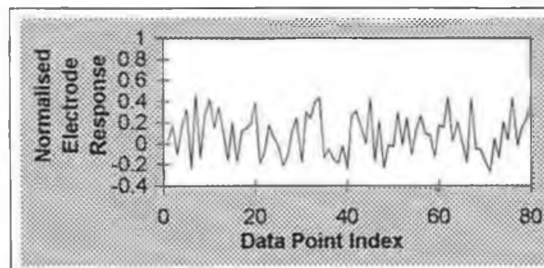


Ca ISE

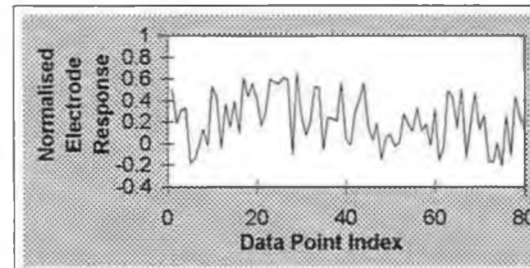
Pattern 26



Na ISE

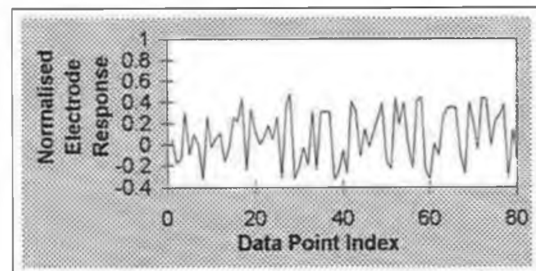


K ISE

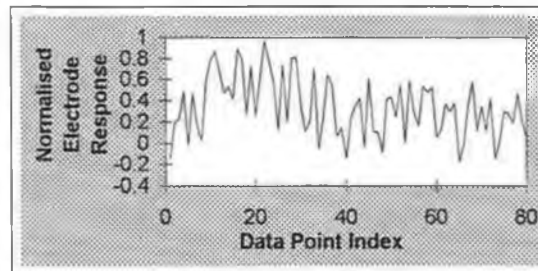


Ca ISE

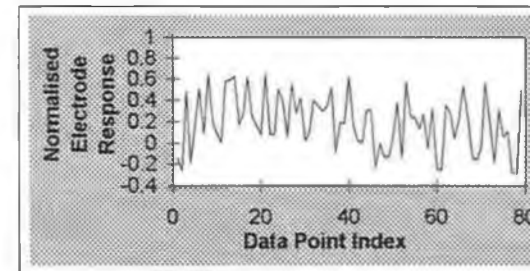
Pattern 27



Na ISE

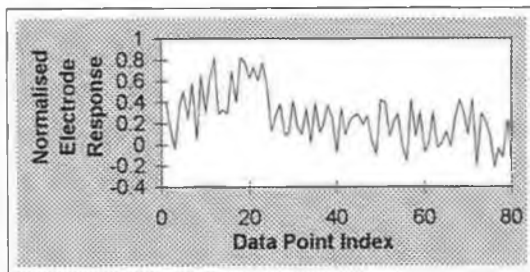


K ISE

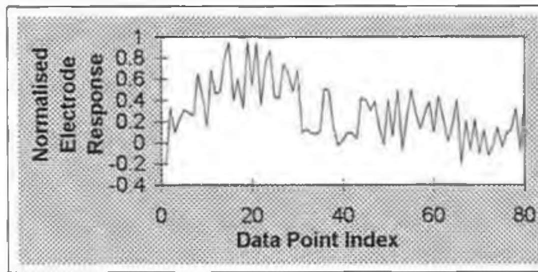


Ca ISE

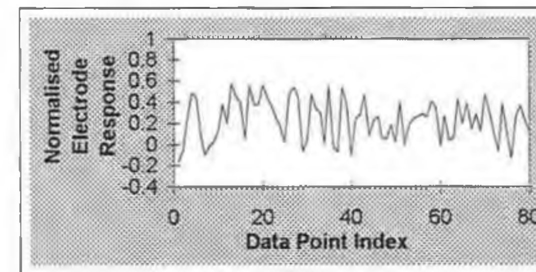
Pattern 28



Na ISE

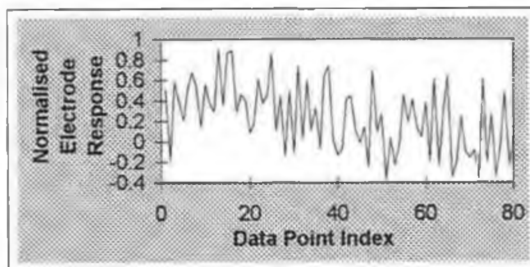


K ISE

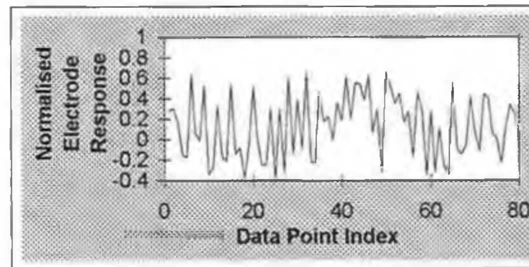


Ca ISE

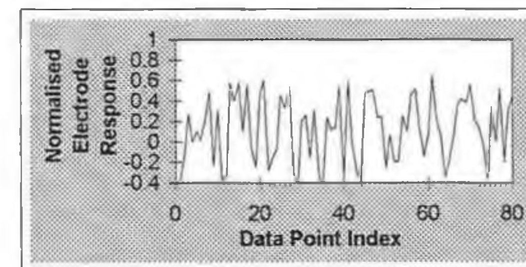
Pattern 29



Na ISE

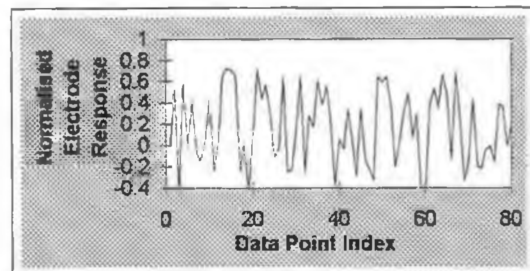


K ISE

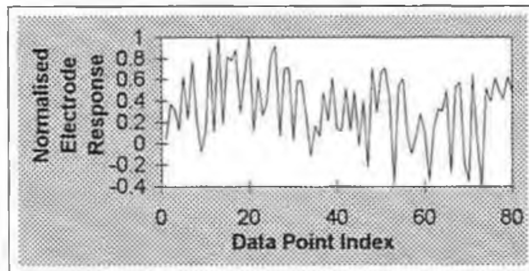


Ca ISE

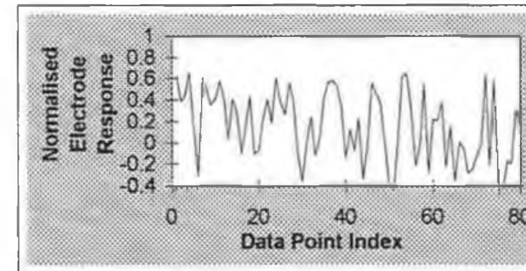
Pattern 30



Na ISE

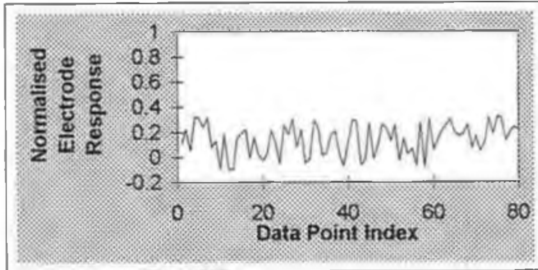


K ISE

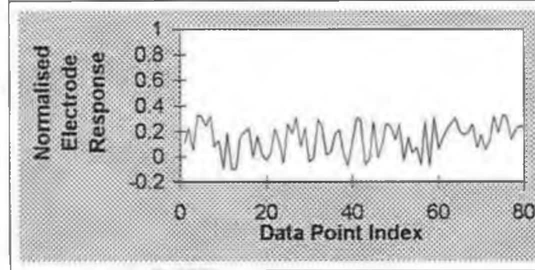


Ca ISE

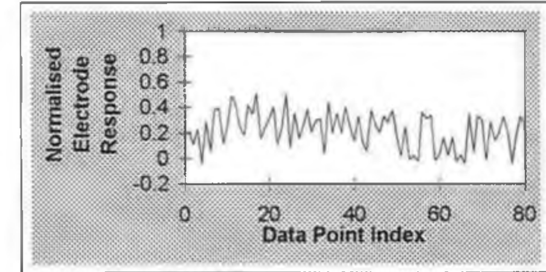
Pattern 31



Na ISE

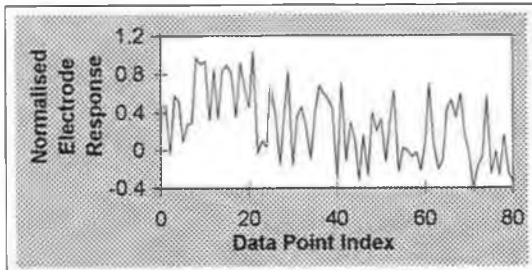


K ISE

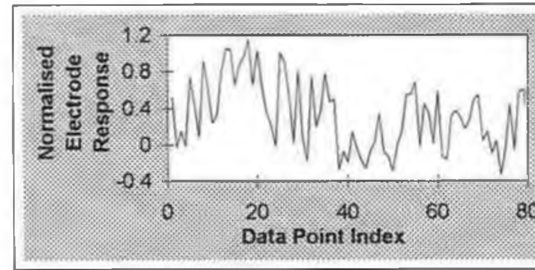


Ca ISE

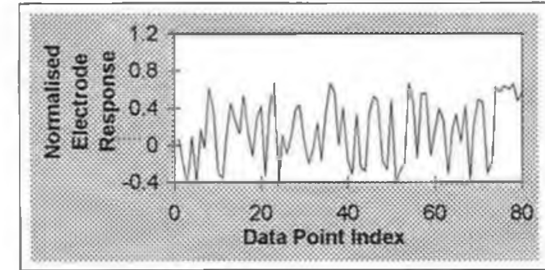
Pattern 32



Na ISE

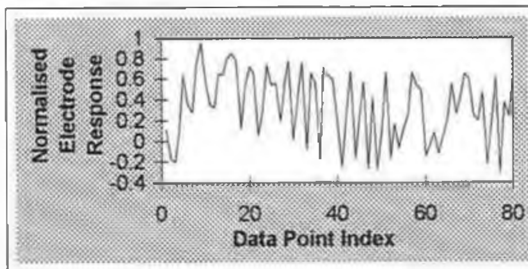


K ISE

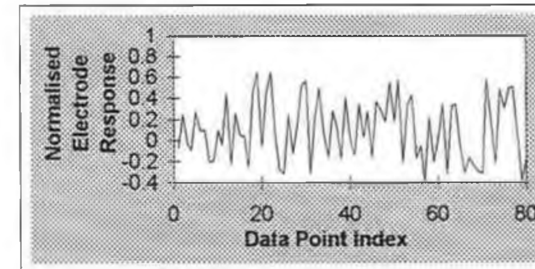


Ca ISE

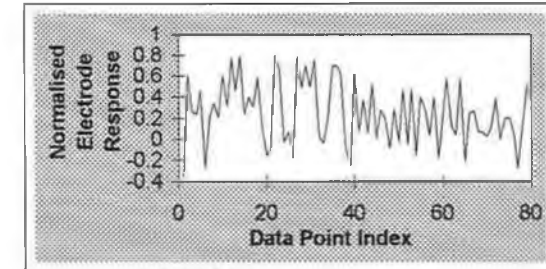
Pattern 33



Na ISE

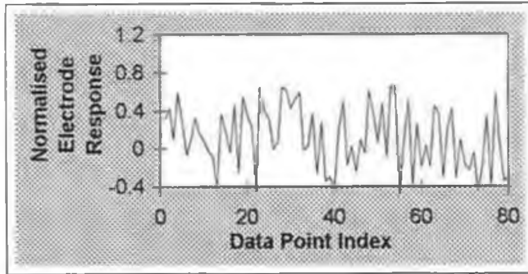


K ISE

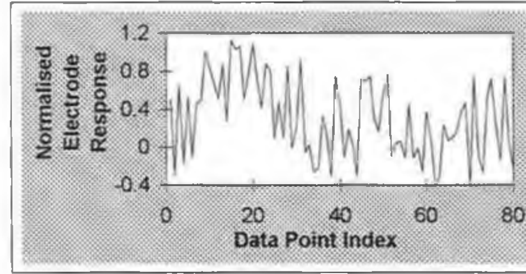


Ca ISE

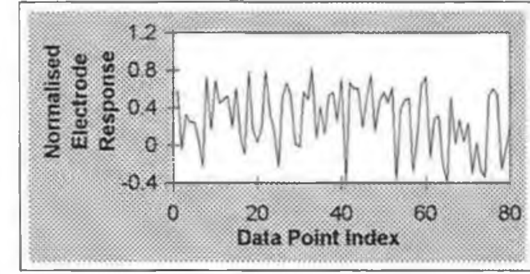
Pattern 34



Na ISE

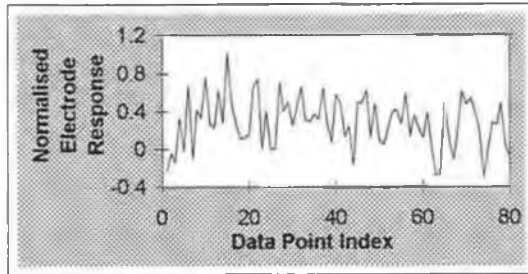


K ISE

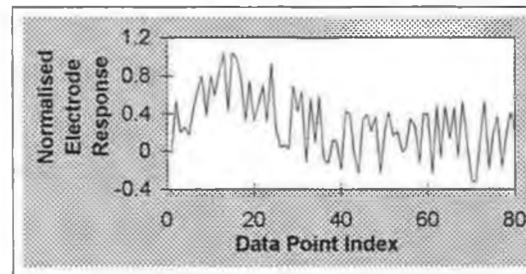


Ca ISE

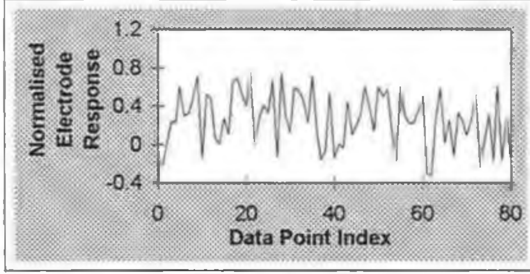
Pattern 35



Na ISE

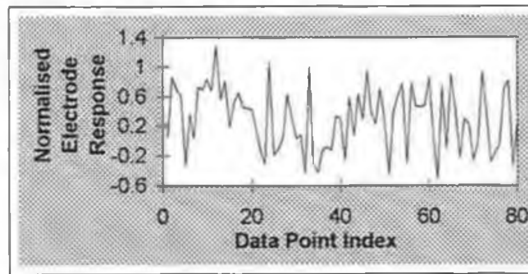


K ISE

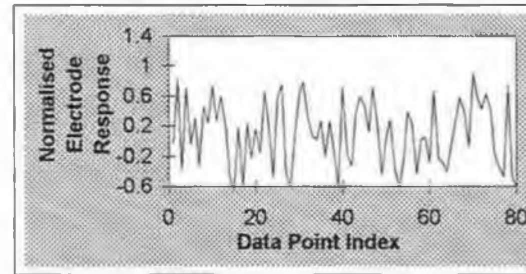


Ca ISE

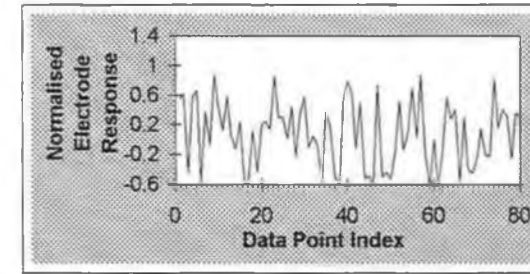
Pattern 36



Na ISE

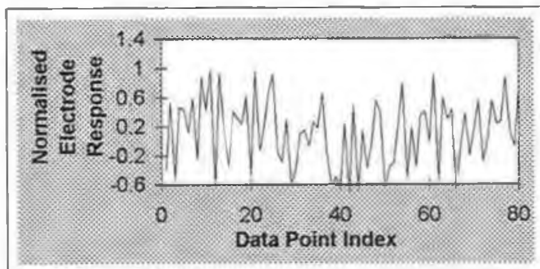


K ISE

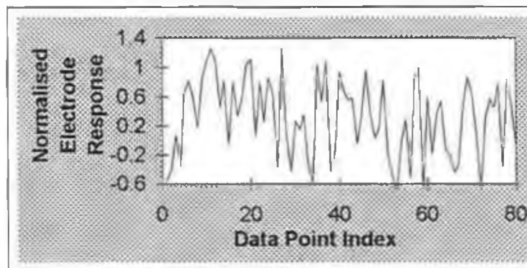


Ca ISE

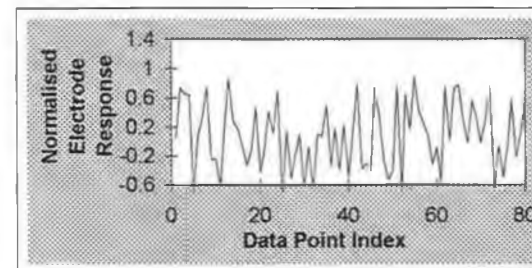
Pattern 37



Na ISE

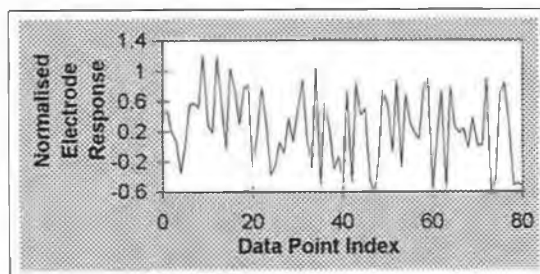


K ISE

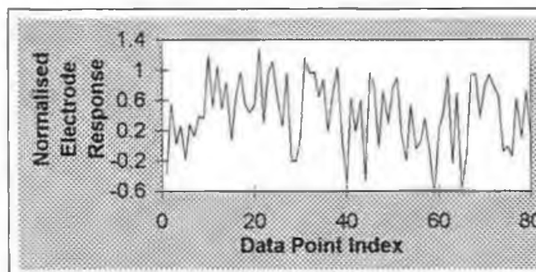


Ca ISE

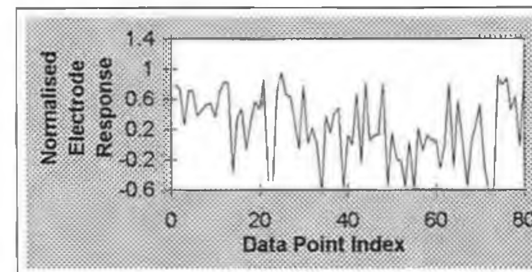
Pattern 38



Na ISE

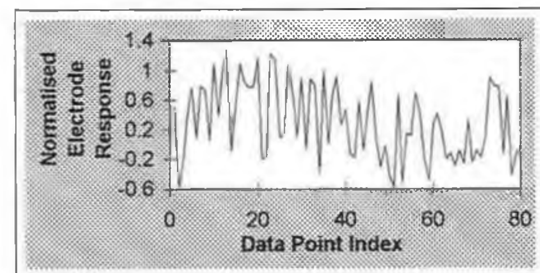


K ISE

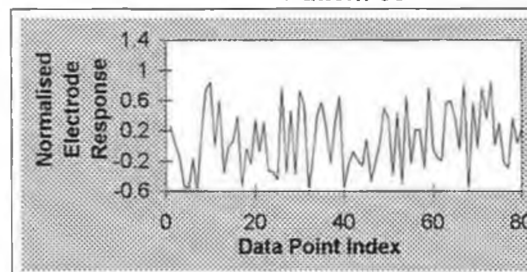


Ca ISE

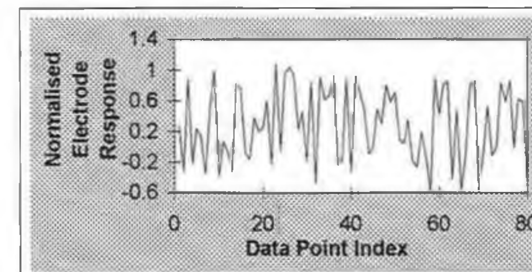
Pattern 39



Na ISE

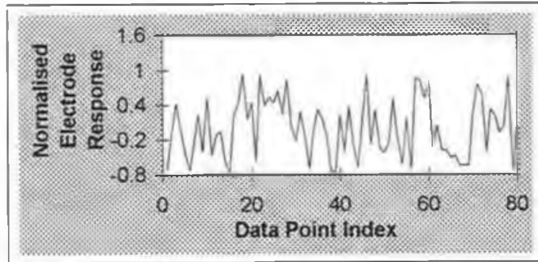


K ISE

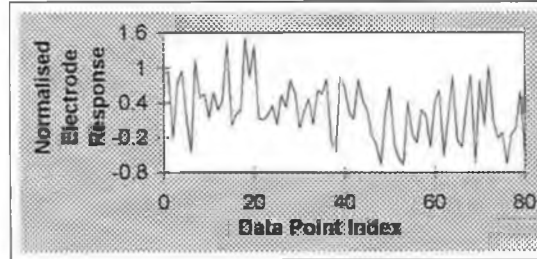


Ca ISE

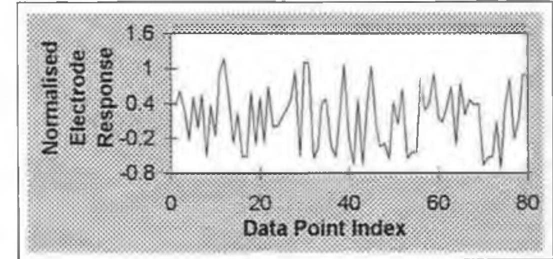
Pattern 40



Na ISE

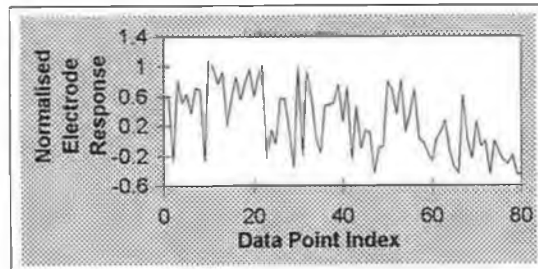


K ISE

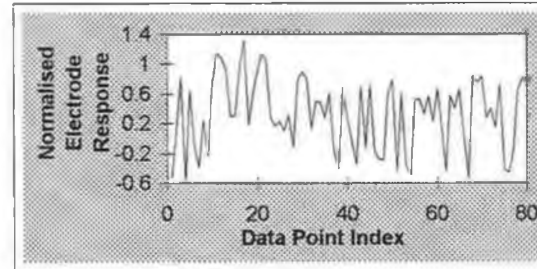


Ca ISE

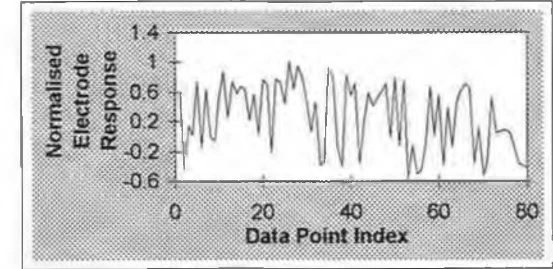
Pattern 41



Na ISE

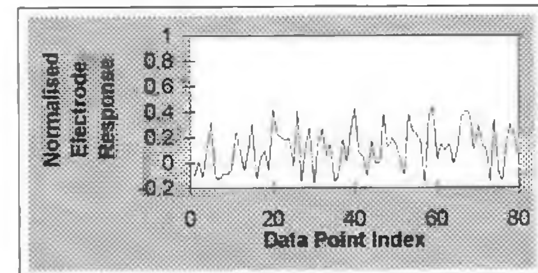


K ISE

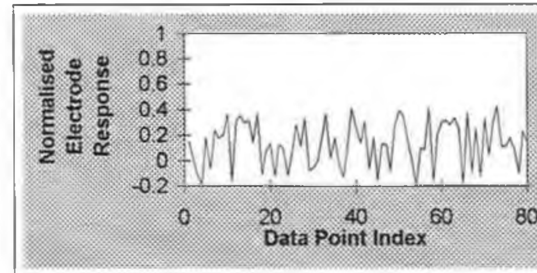


Ca ISE

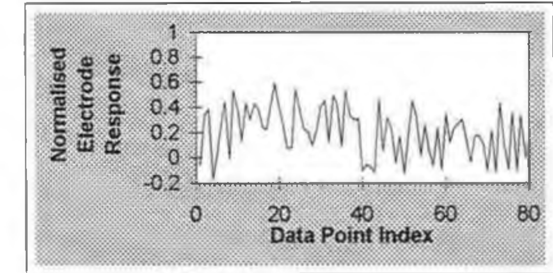
Pattern 42



Na ISE

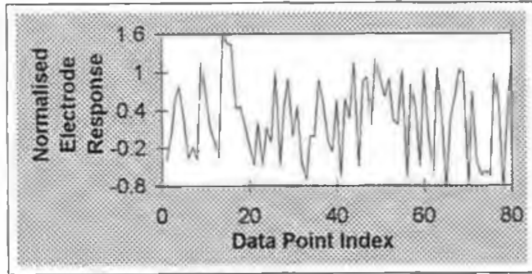


K ISE

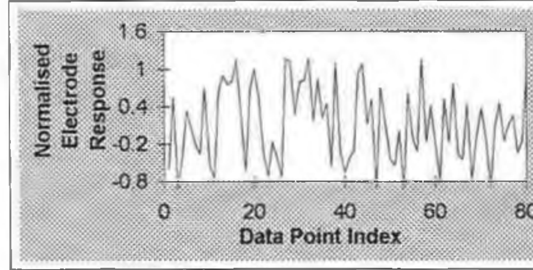


Ca ISE

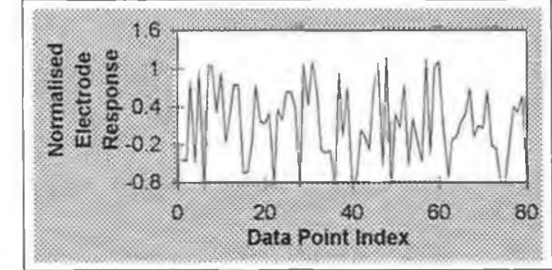
Pattern 43



Na ISE

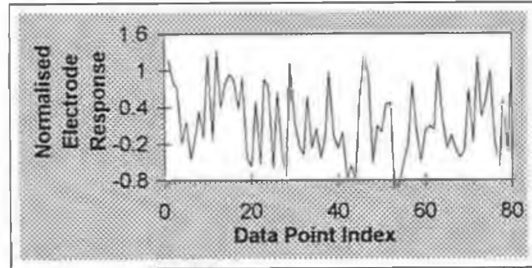


K ISE

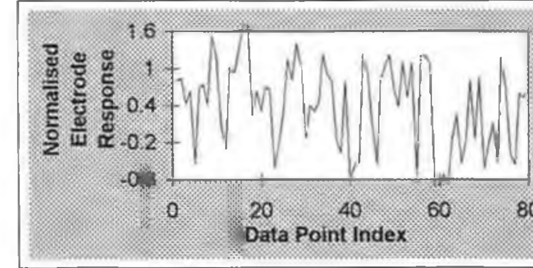


Ca ISE

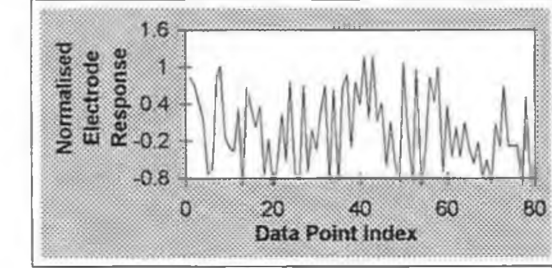
Pattern 44



Na ISE

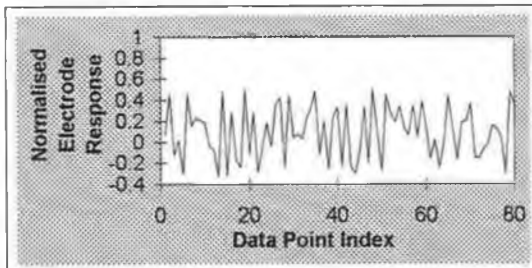


K ISE

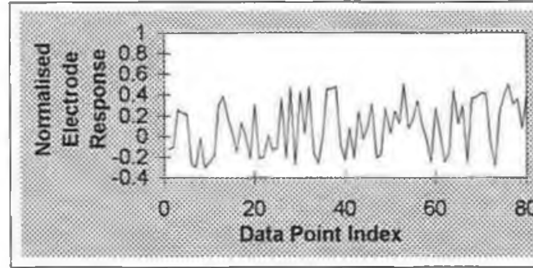


Ca ISE

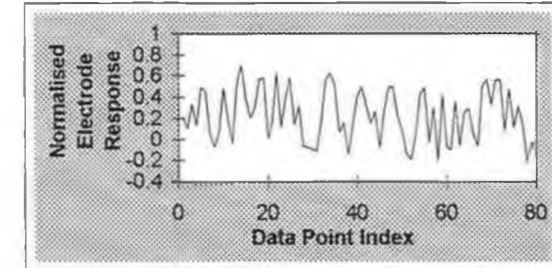
Pattern 45



Na ISE

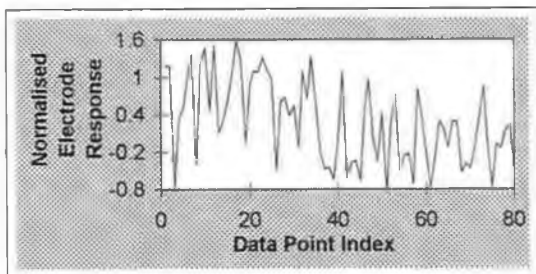


K ISE

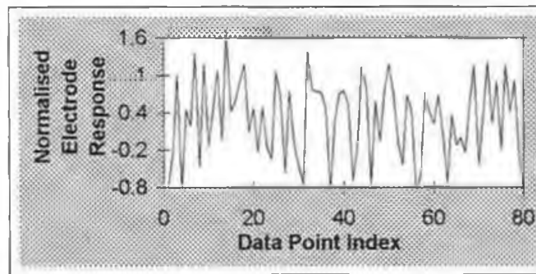


Ca ISE

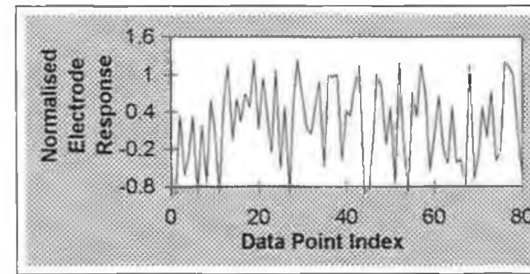
Pattern 46



Na ISE

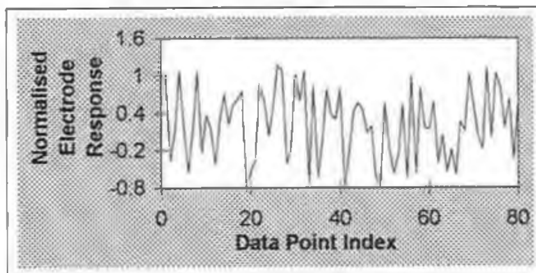


K ISE

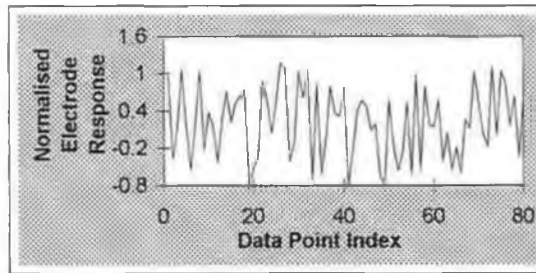


Ca ISE

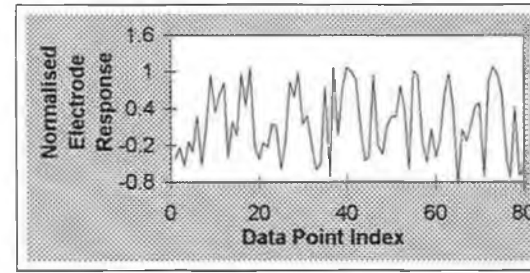
Pattern 47



Na ISE

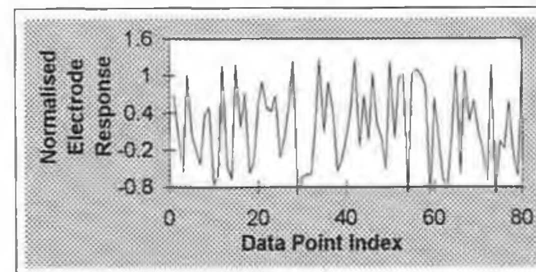


K ISE

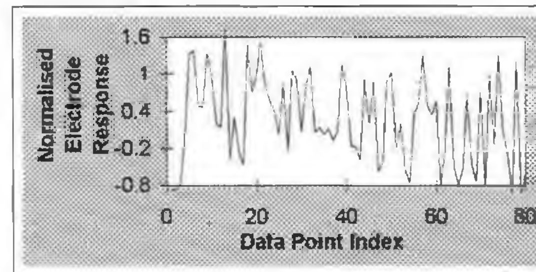


Ca ISE

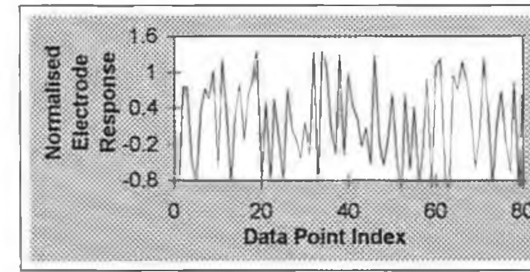
Pattern 48



Na ISE

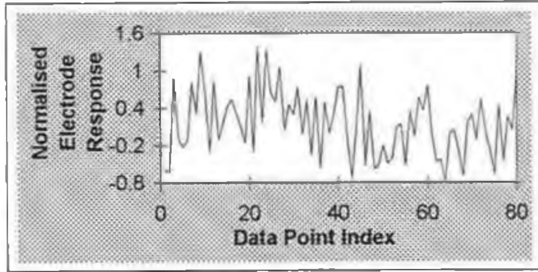


K ISE

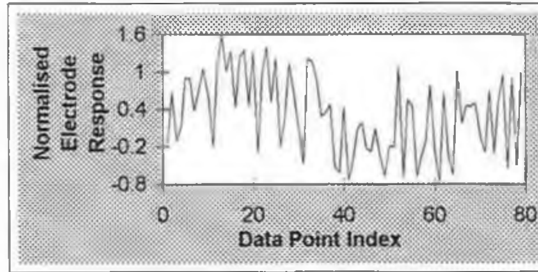


Ca ISE

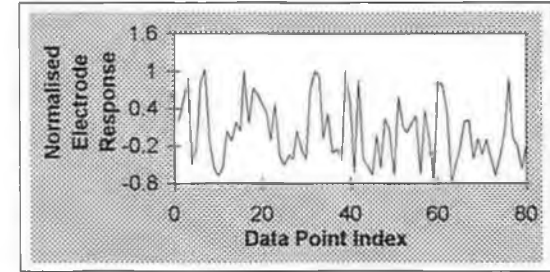
Pattern 49



Na ISE



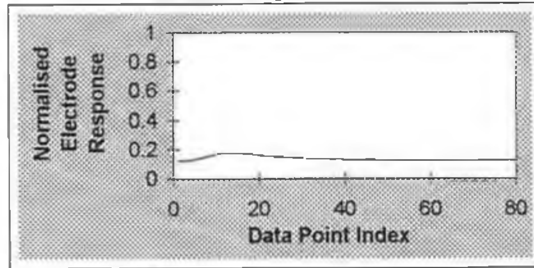
K ISE



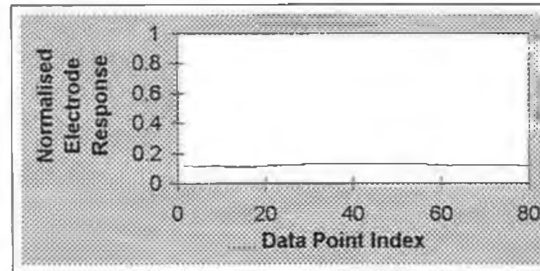
Ca ISE

Appendix 3 Test Patterns Used For The Study Of The Effect Of Peak Height Variation On Pattern Classification In Chapter 2
[See 2.3.1.4 and 2.3.2.2]

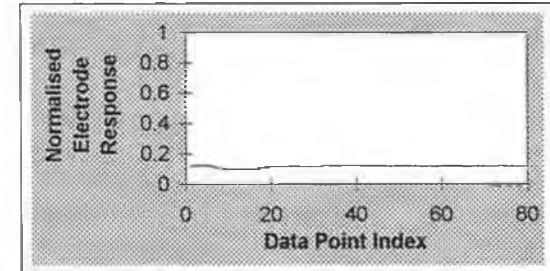
Pattern 1



Na

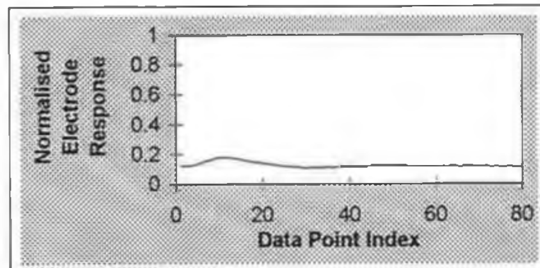


K ISE

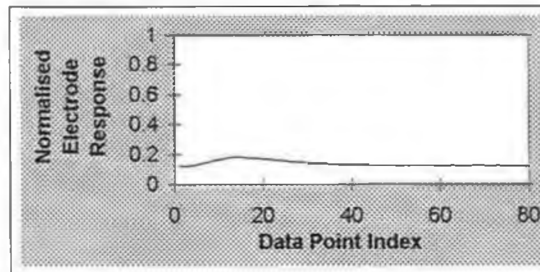


Ca ISE

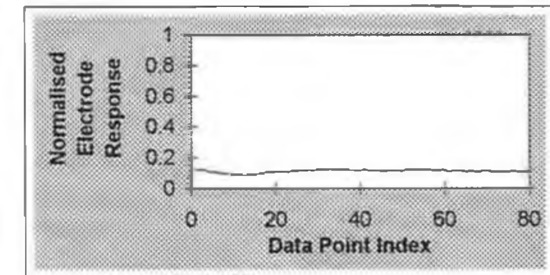
Pattern 2



Na ISE

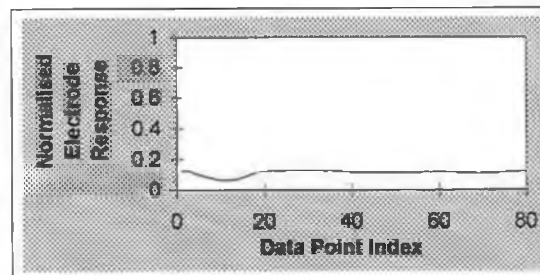


K ISE

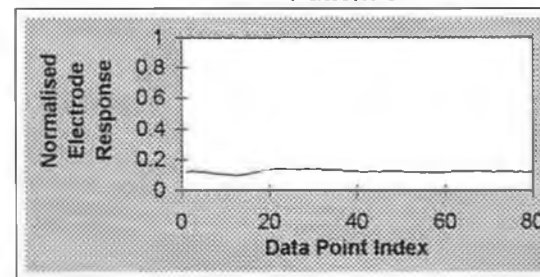


Ca ISE

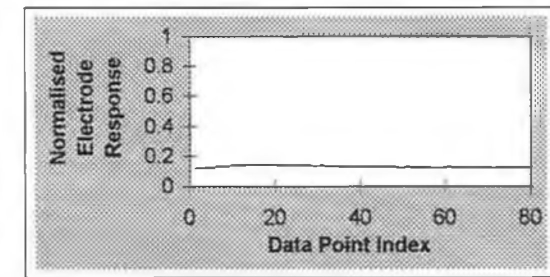
Pattern 3



Na ISE

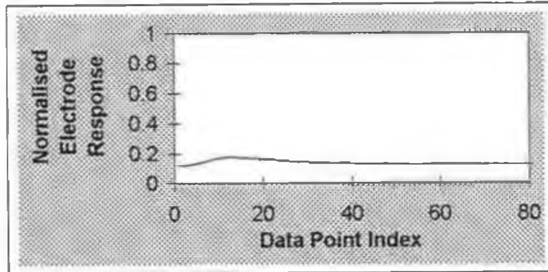


K ISE

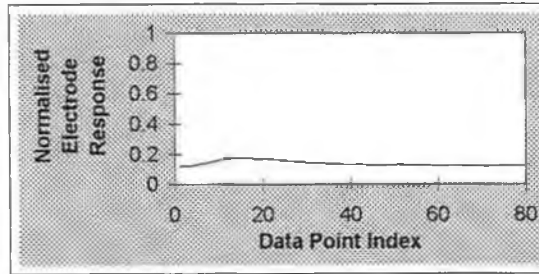


Ca ISE

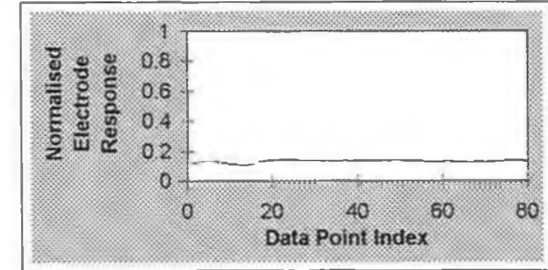
Pattern 4



Na ISE

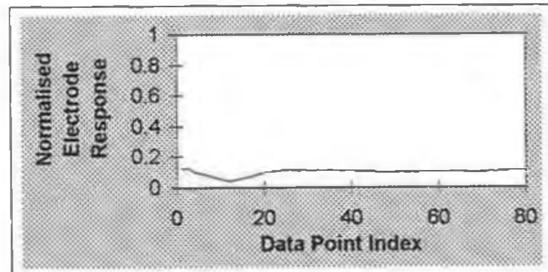


K ISE

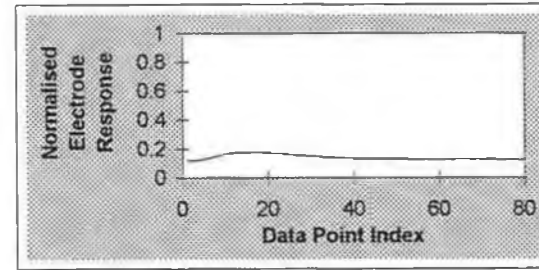


Ca ISE

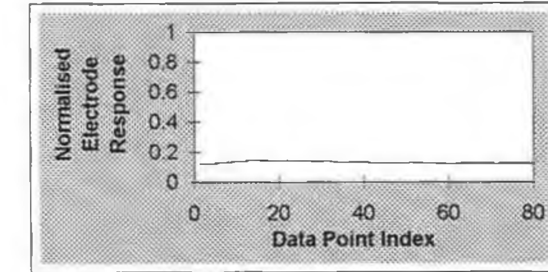
Pattern 5



Na ISE

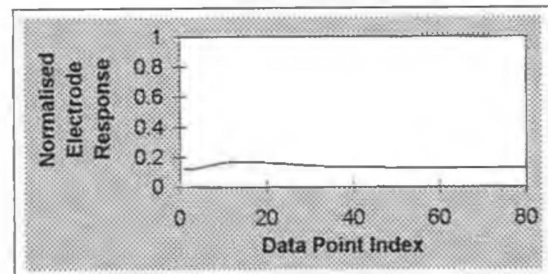


K ISE

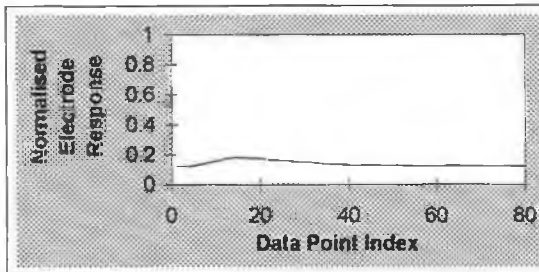


Ca ISE

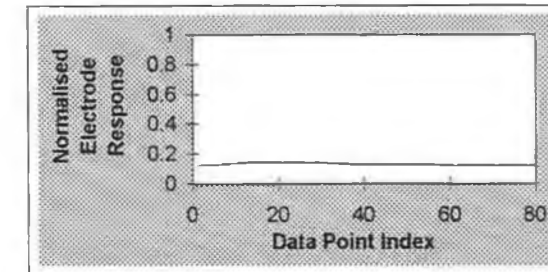
Pattern 6



Na ISE

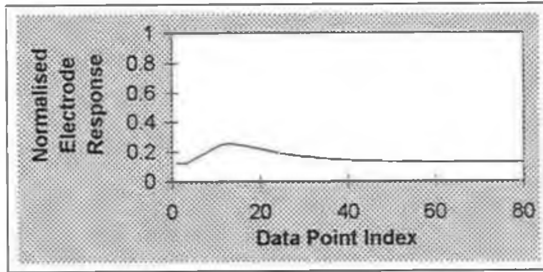


K ISE

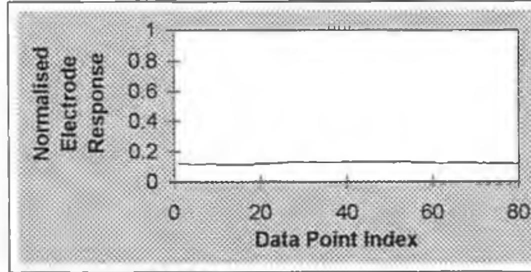


Ca ISE

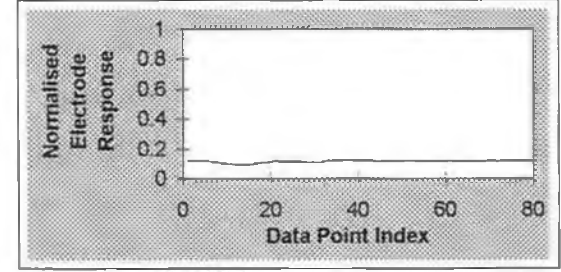
Pattern 7



Na ISE

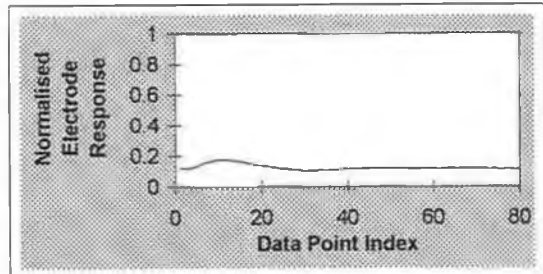


K ISE

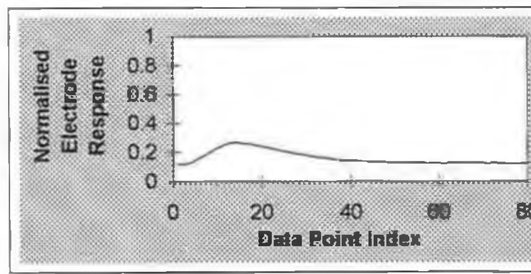


Ca ISE

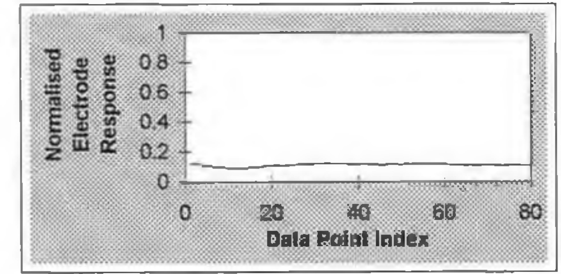
Pattern 8



Na ISE

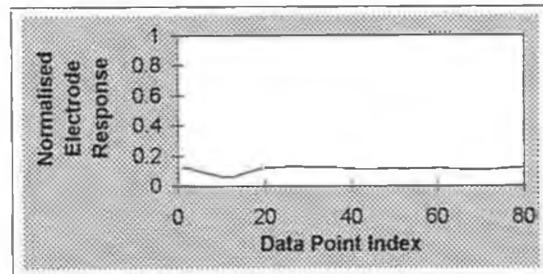


K ISE

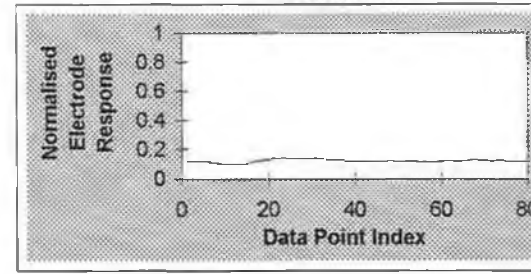


Ca ISE

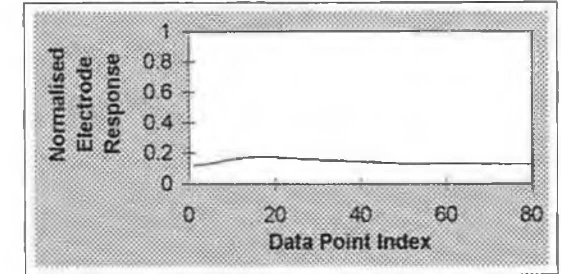
Pattern 9



Na ISE

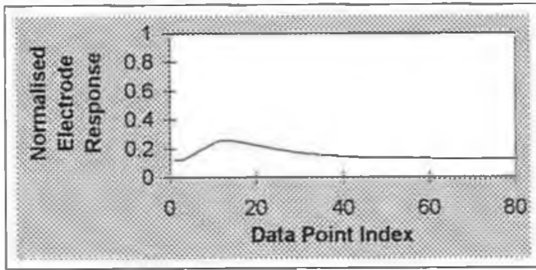


K ISE

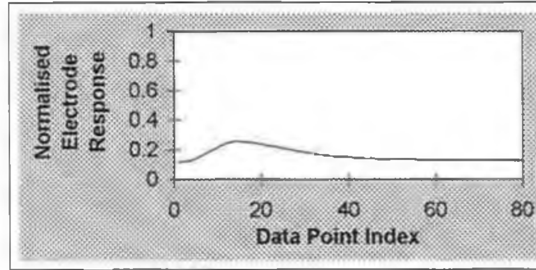


Ca ISE

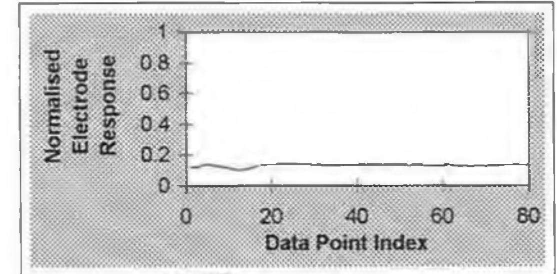
Pattern 10



Na ISE

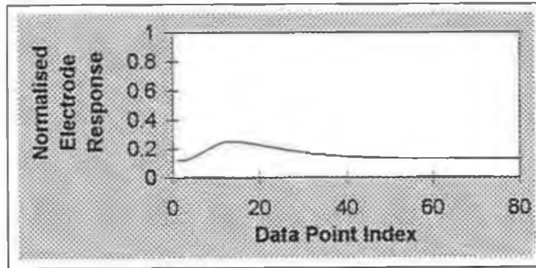


K ISE

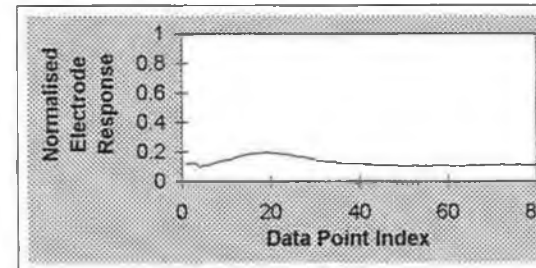


Ca ISE

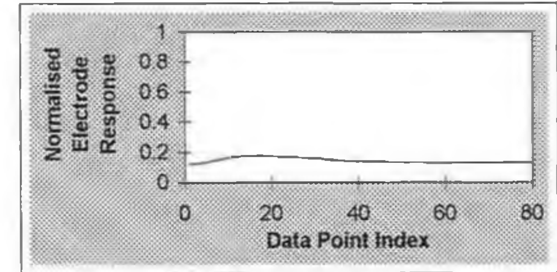
Pattern 11



Na ISE

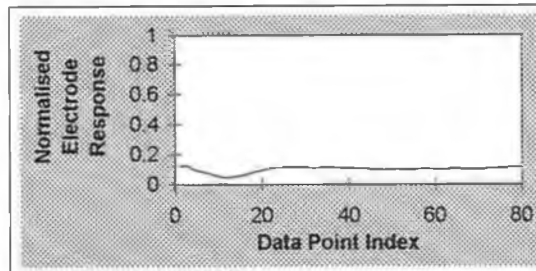


K ISE

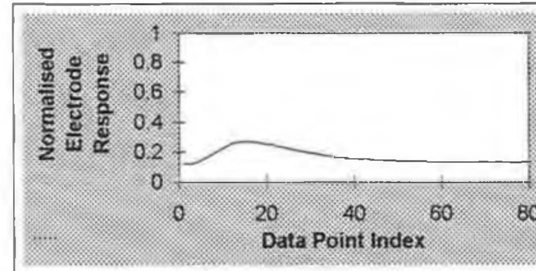


Ca ISE

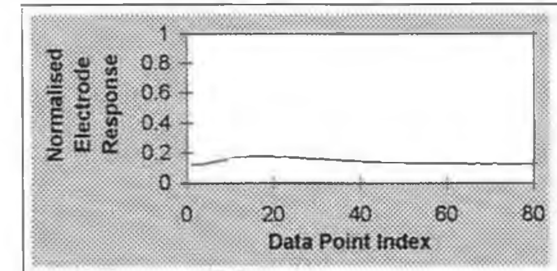
Pattern 12



Na ISE

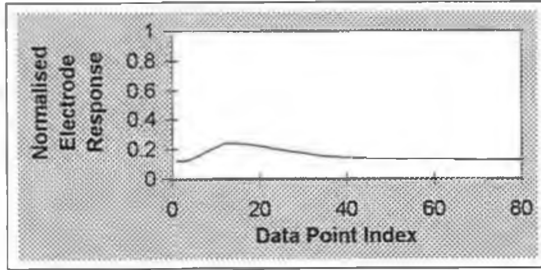


K ISE

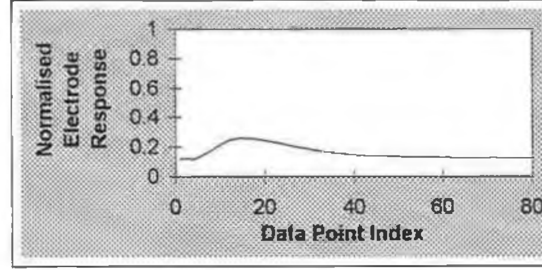


Ca ISE

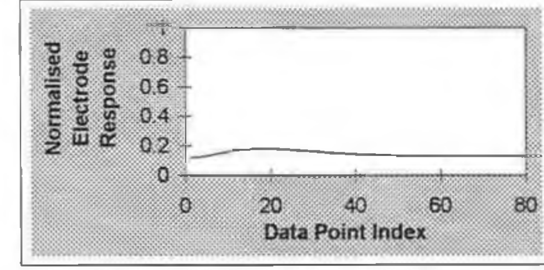
Pattern 13



Na ISE

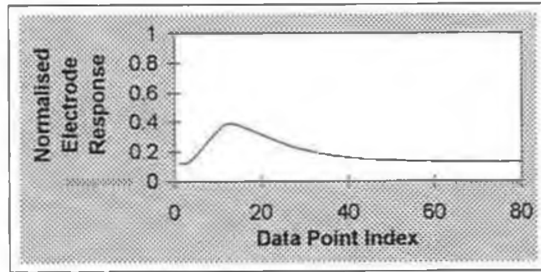


K ISE

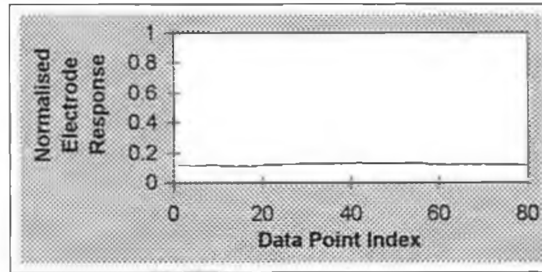


Ca ISE

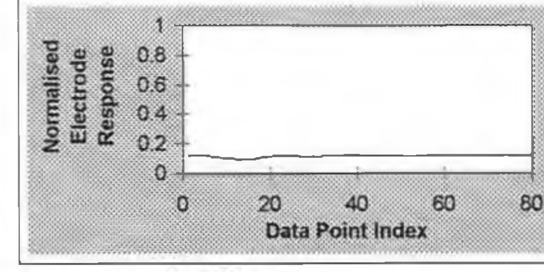
Pattern 14



Na ISE

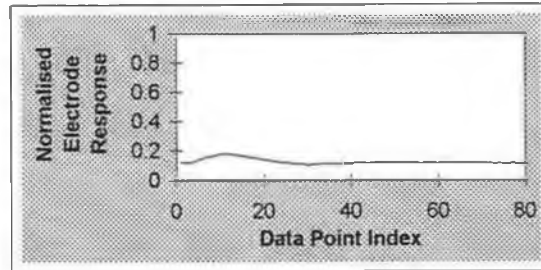


K ISE

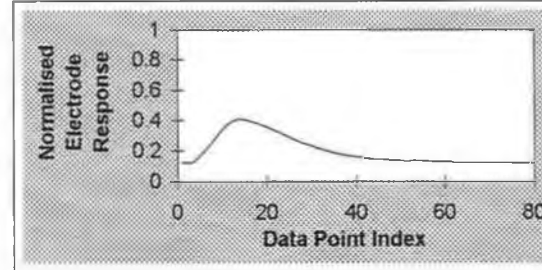


Ca ISE

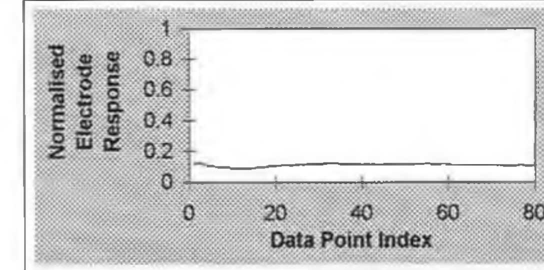
Pattern 15



Na ISE

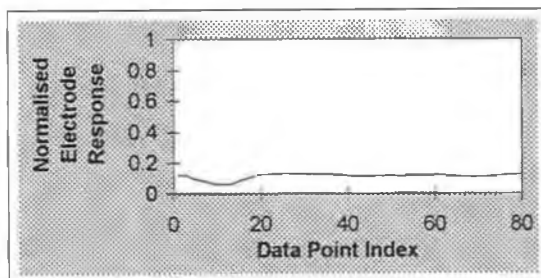


K ISE

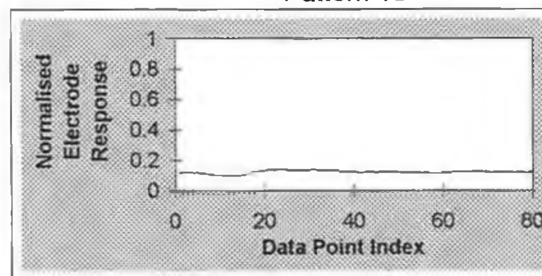


Ca ISE

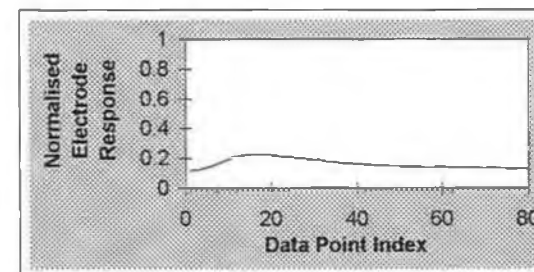
Pattern 16



Na ISE

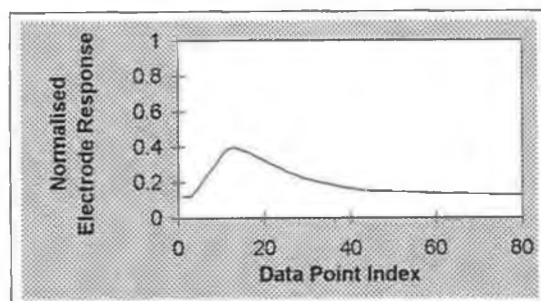


K ISE

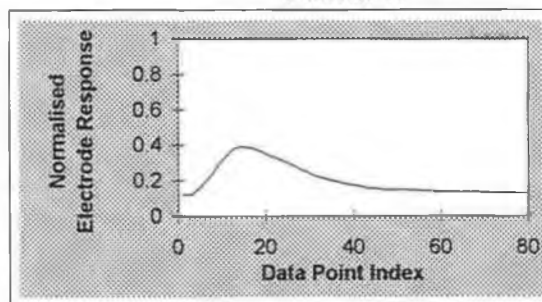


Ca ISE

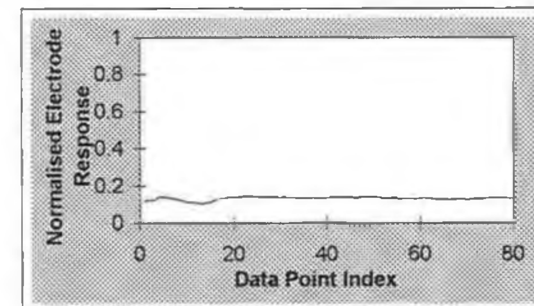
Pattern 17



Na ISE

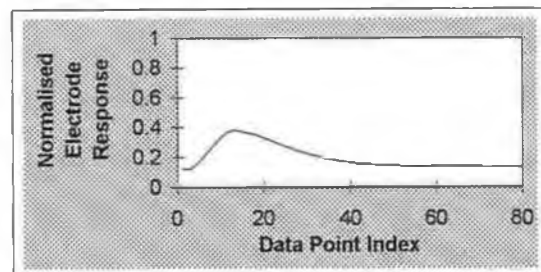


K ISE

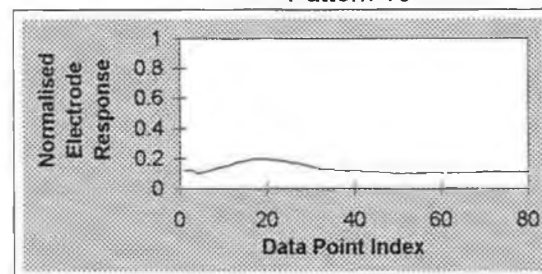


Ca ISE

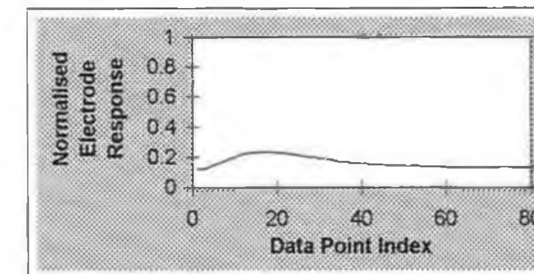
Pattern 18



Na ISE

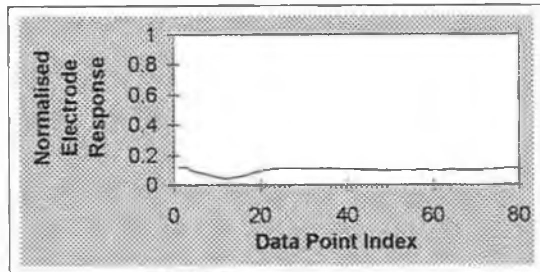


K ISE

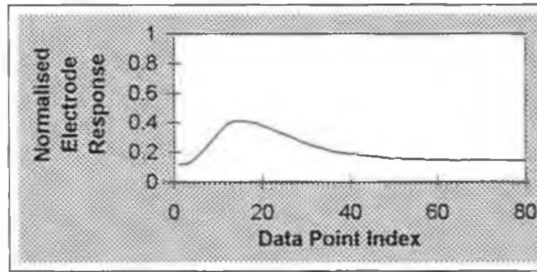


Ca ISE

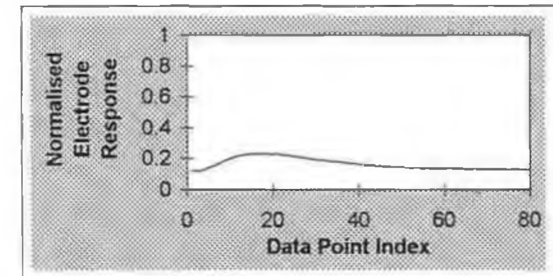
Pattern 19



Na ISE

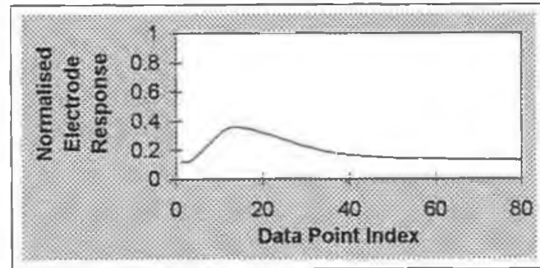


K ISE

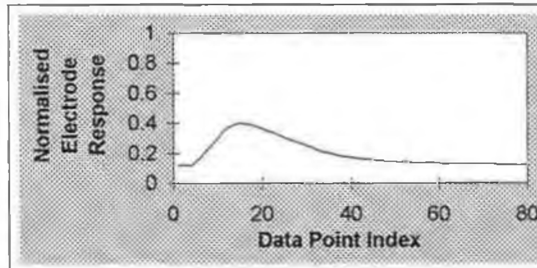


Ca ISE

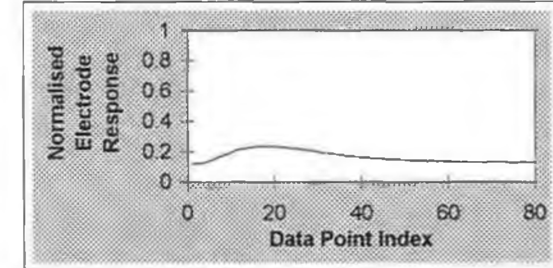
Pattern 20



Na ISE

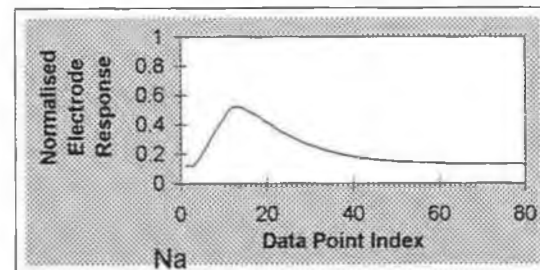


K ISE

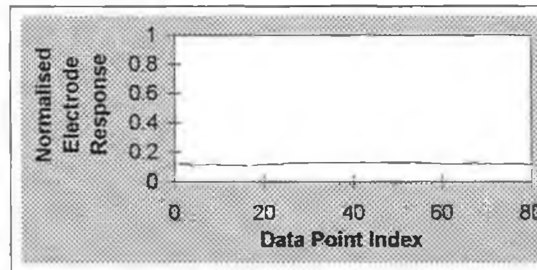


Ca ISE

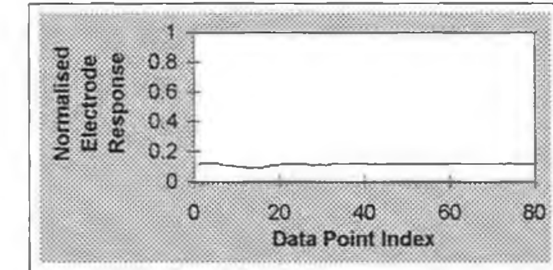
Pattern 21



Na ISE

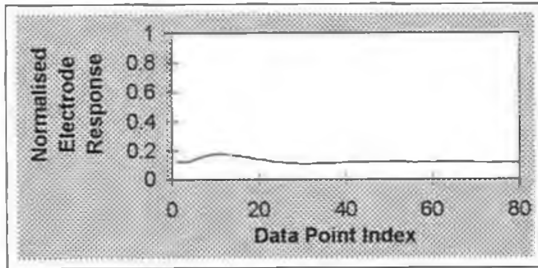


K ISE

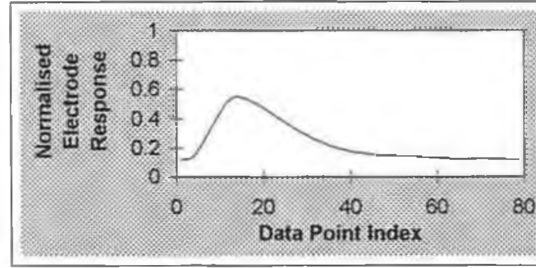


Ca ISE

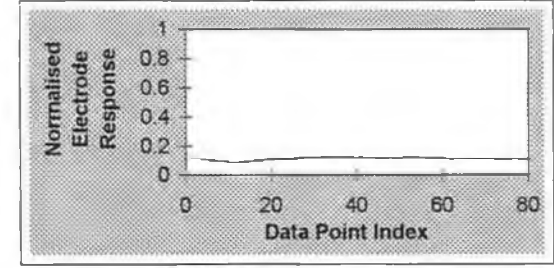
Pattern 22



Na ISE

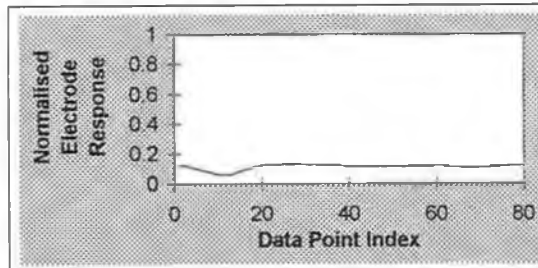


K ISE

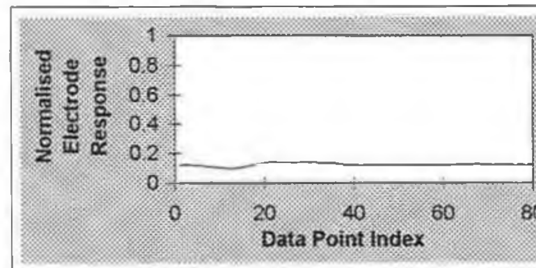


Ca ISE

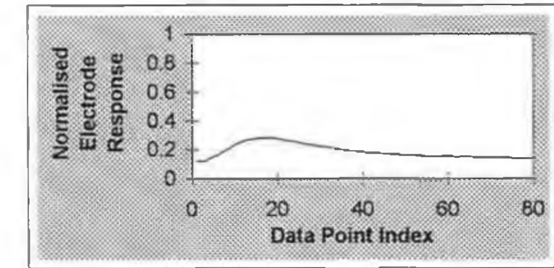
Pattern 23



Na ISE

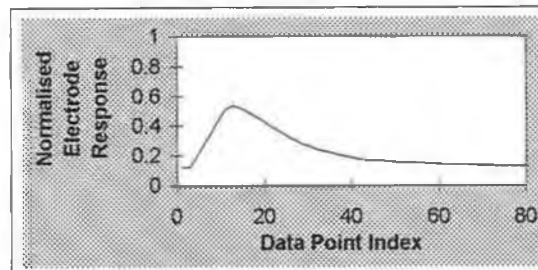


K ISE

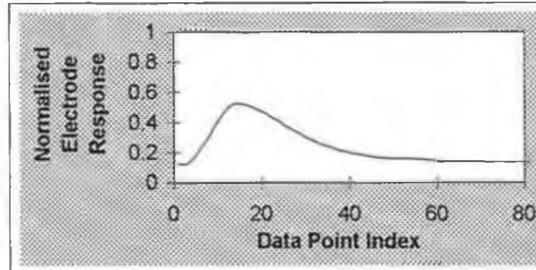


Ca ISE

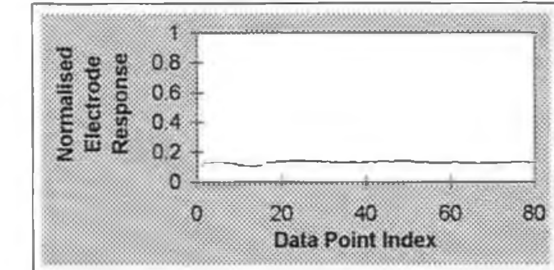
Pattern 24



Na ISE

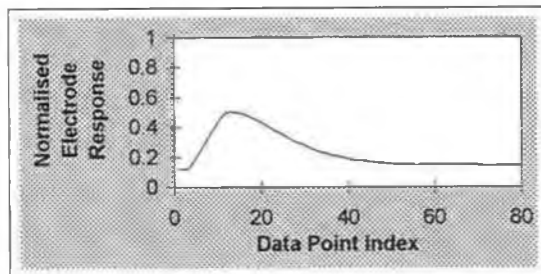


K ISE

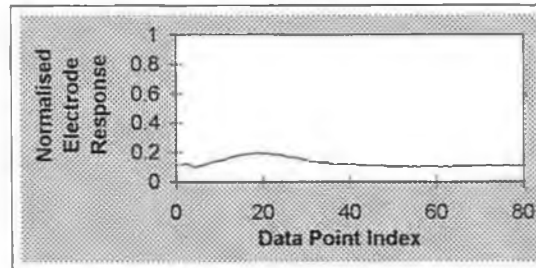


Ca ISE

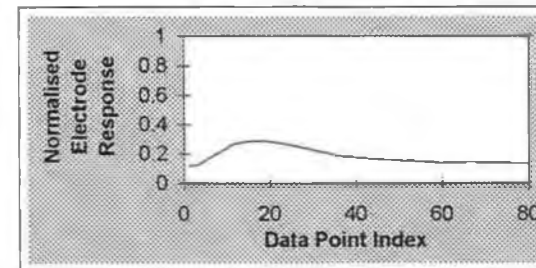
Pattern 25



Na ISE

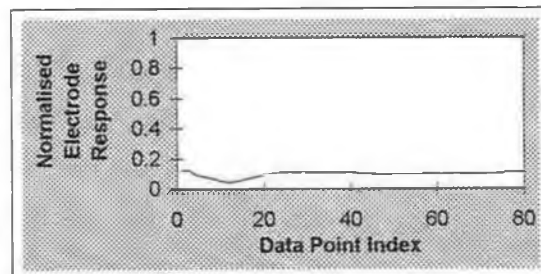


K ISE

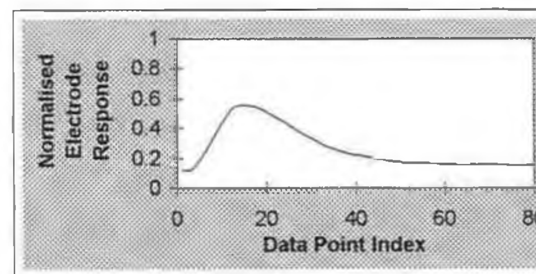


Ca ISE

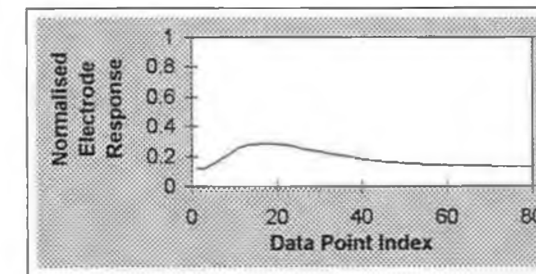
Pattern 26



Na ISE

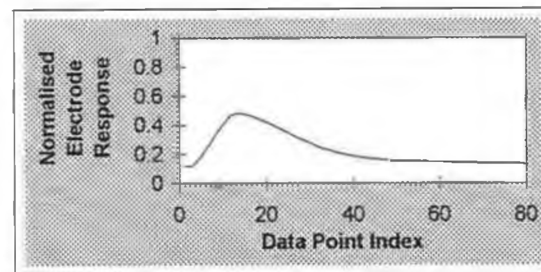


K ISE

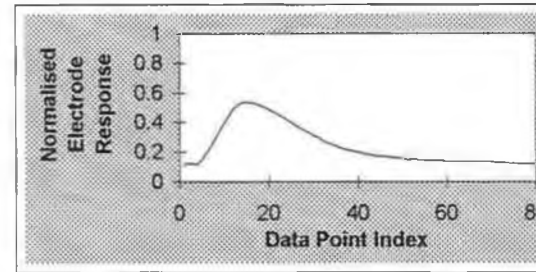


Ca ISE

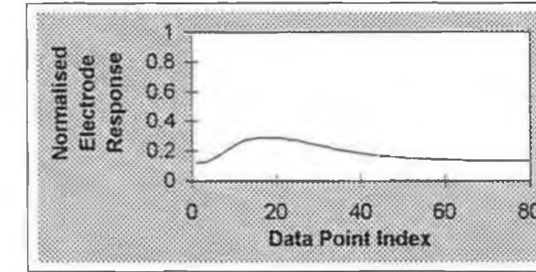
Pattern 27



Na ISE

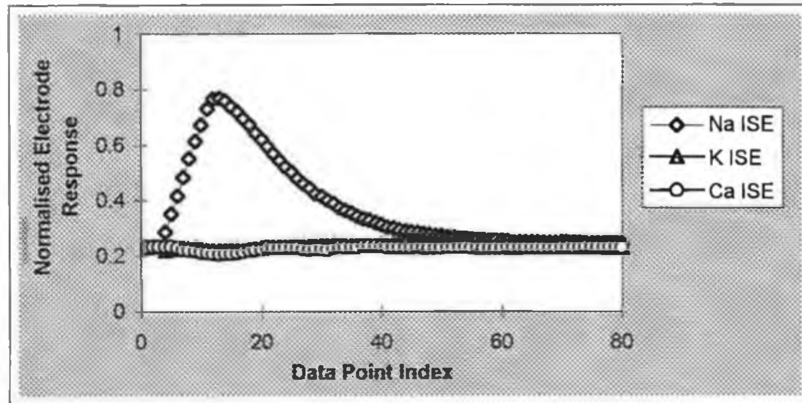


K ISE

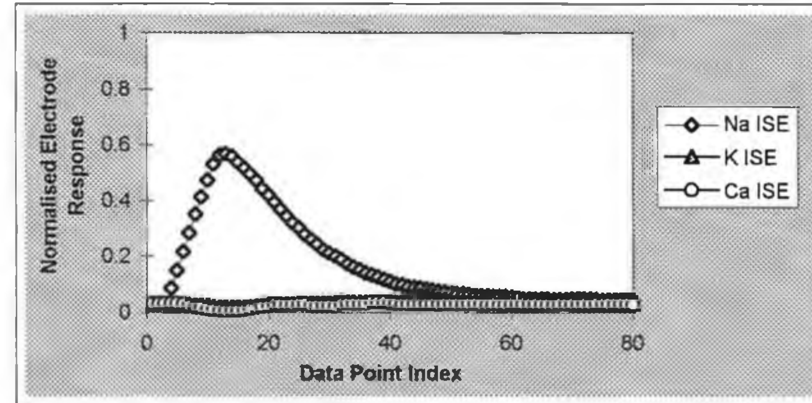


Ca ISE

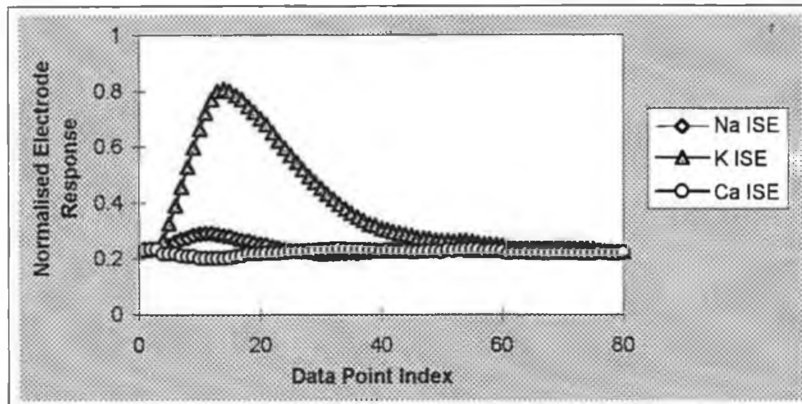
Appendix 4 Test Patterns Used To Study The Influence Of Baseline Shifting On Pattern Classification In Chapter 2 [See 2.3.1.5 and 2.3.2.2.]



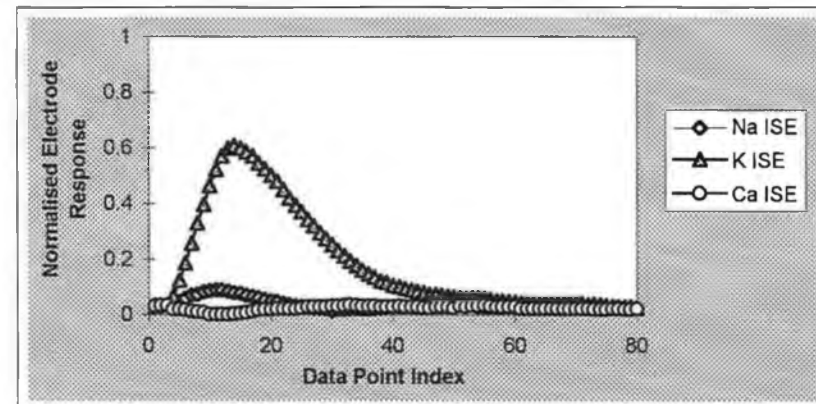
Pattern 1



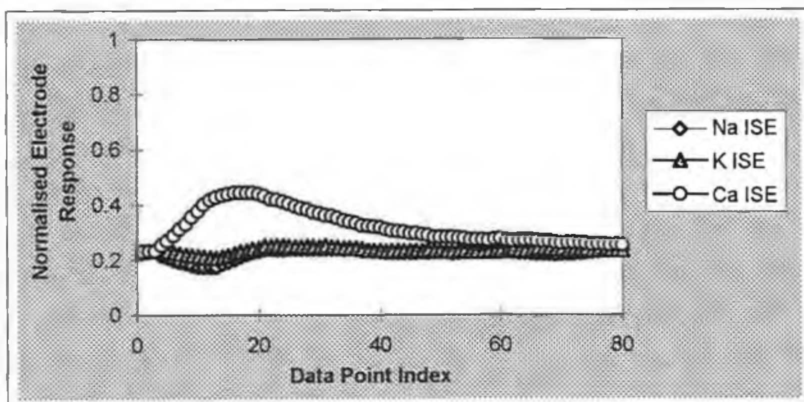
Pattern 2



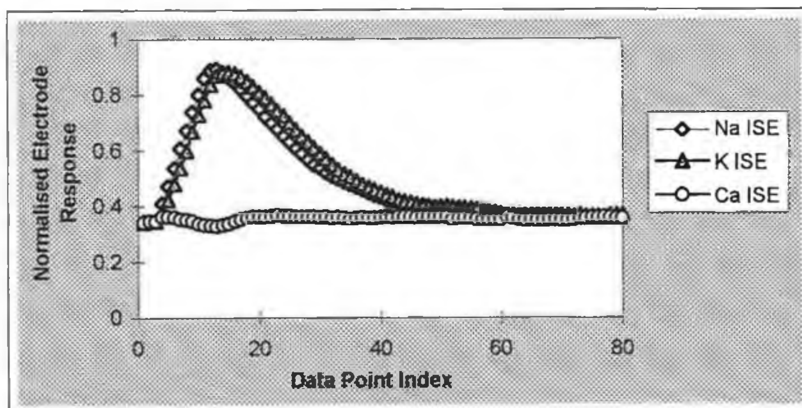
Pattern 3



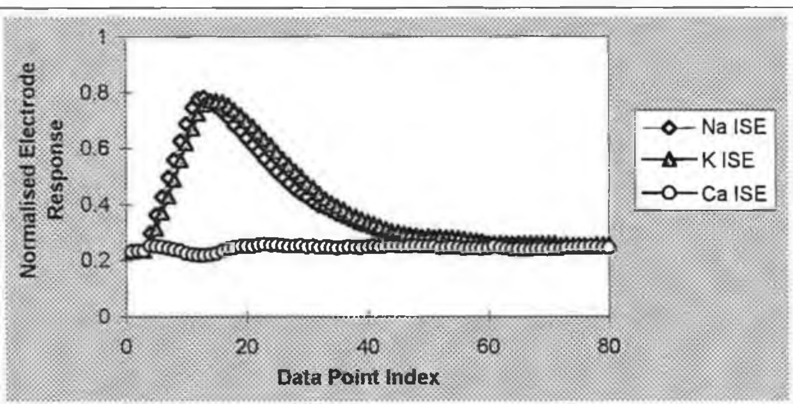
Pattern 4



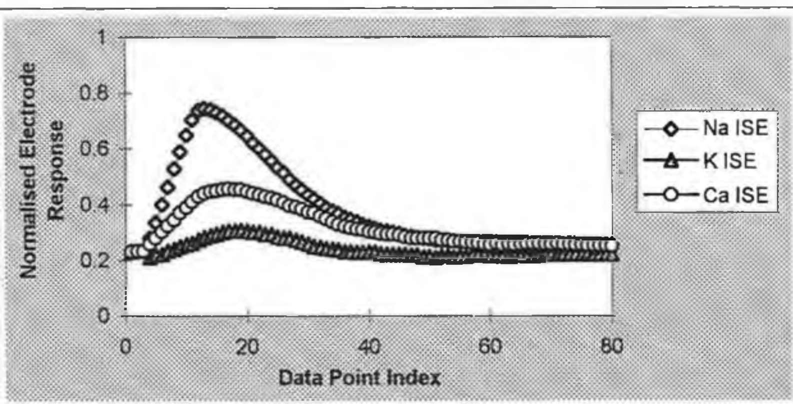
Pattern 5



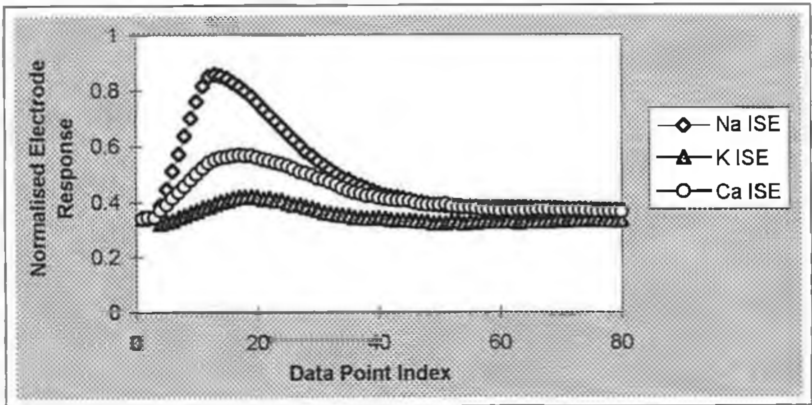
Pattern 7



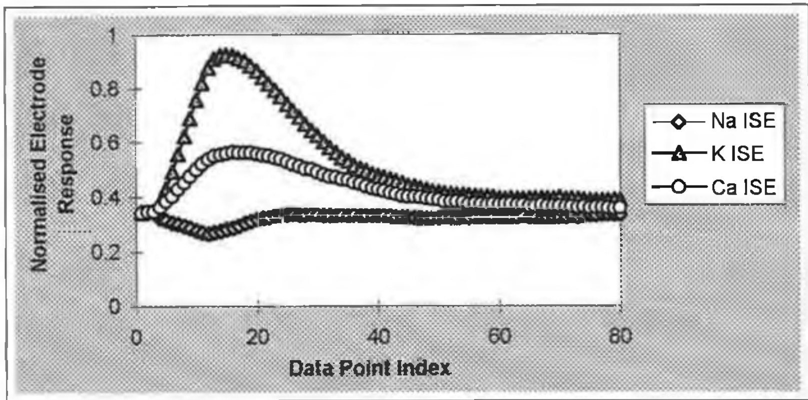
Pattern 6



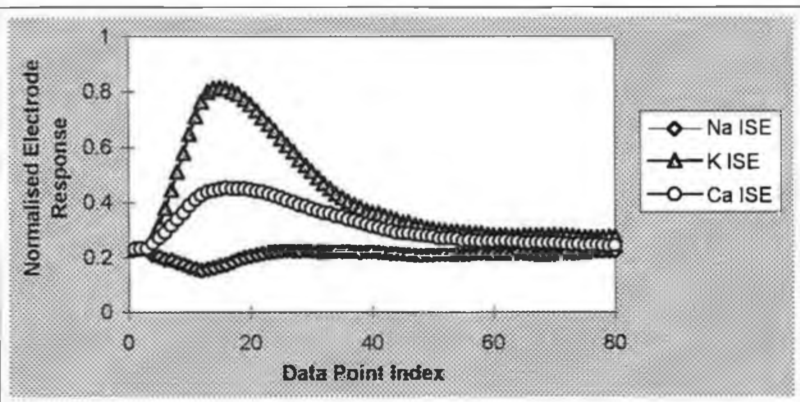
Pattern 8



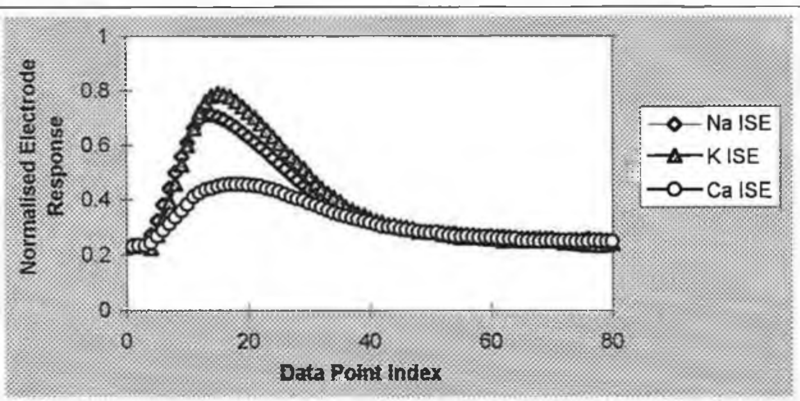
Pattern 9



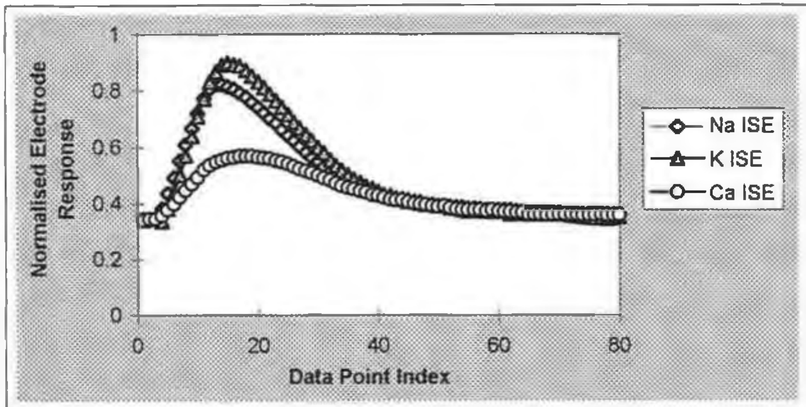
Pattern 11



Pattern 10



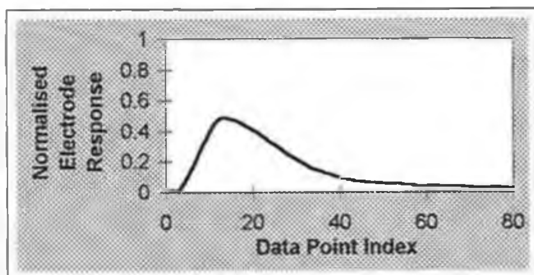
Pattern 12



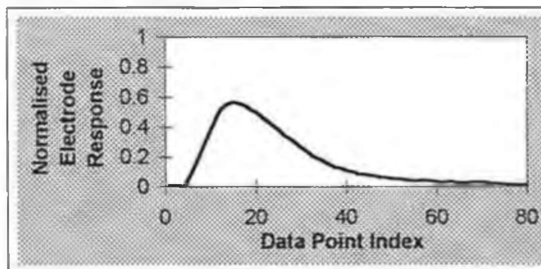
Pattern 13

Appendix 5 Distorted Patterns Used For Broad Training In Chapter 2 [See 2.3.2]

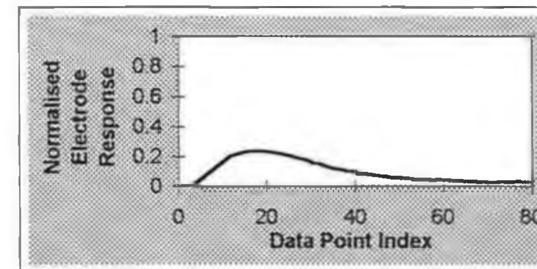
Pattern 1



Na ISE

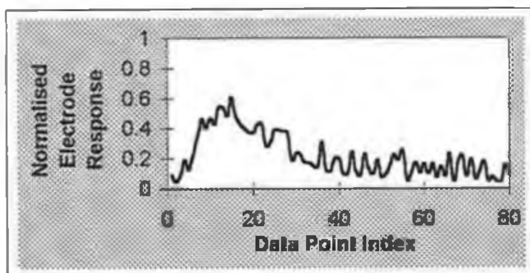


K ISE

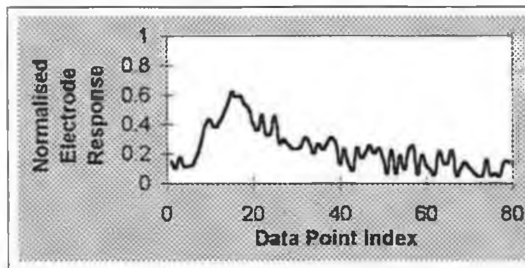


Ca ISE

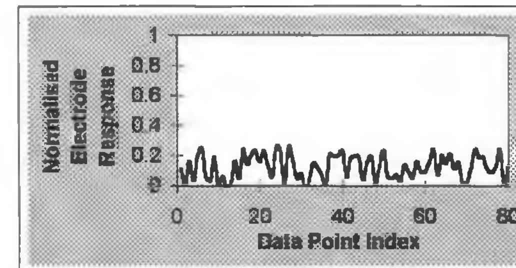
Pattern 2



Na ISE

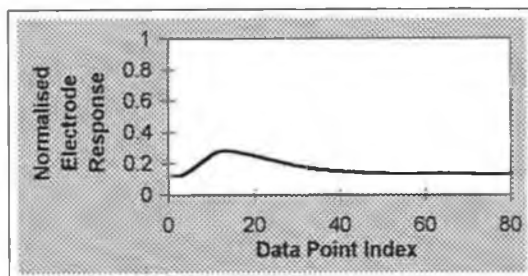


K ISE

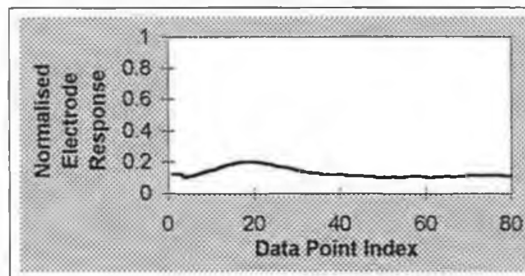


Ca ISE

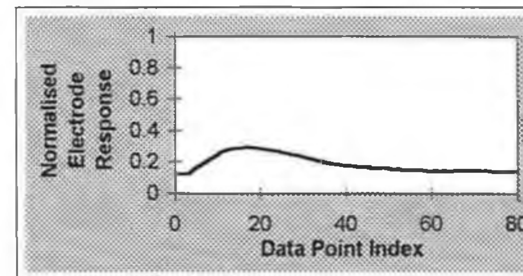
Pattern 3



Na ISE

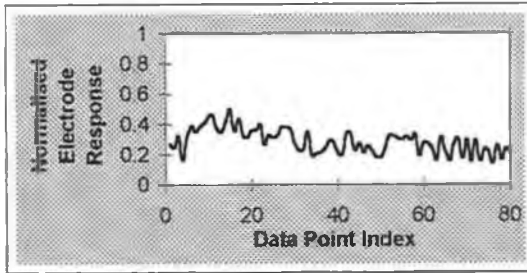


K ISE

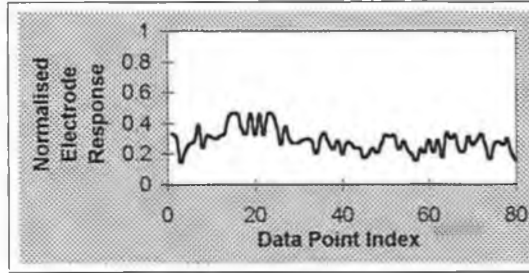


Ca ISE

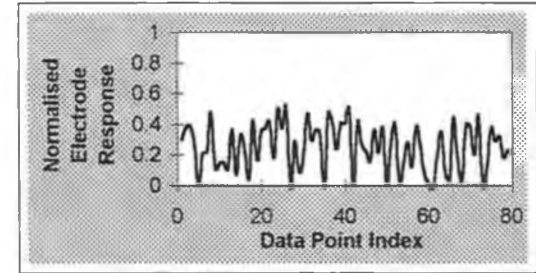
Pattern 4



Na ISE

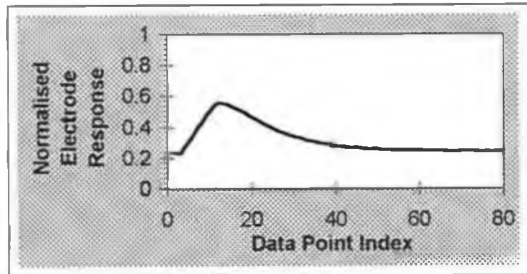


K ISE

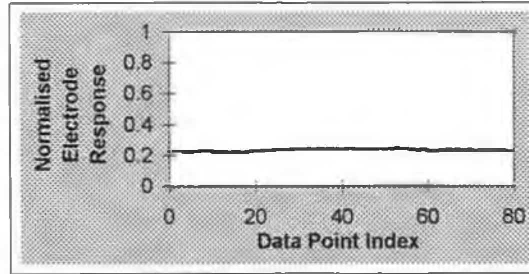


Ca ISE

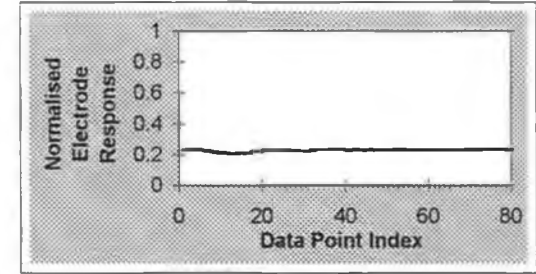
Pattern 5



Na ISE

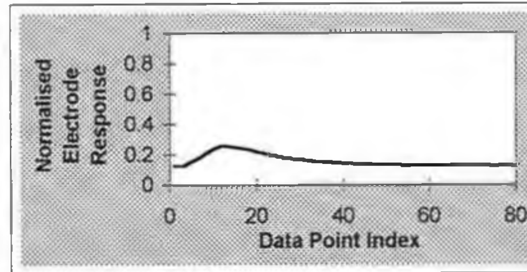


K ISE

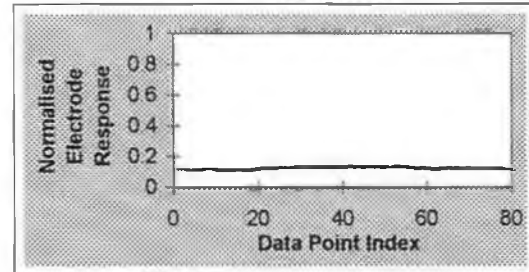


Ca ISE

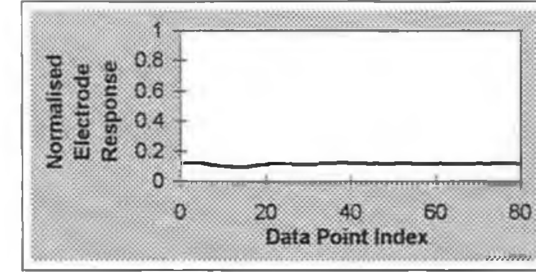
Pattern 6



Na ISE

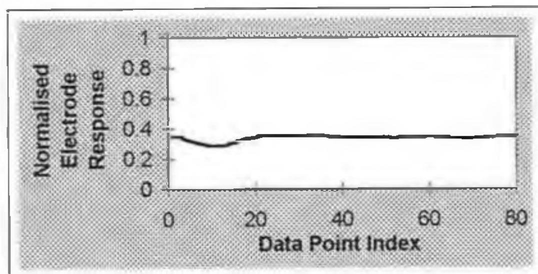


K ISE

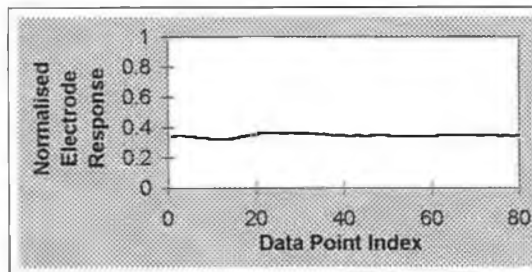


Ca ISE

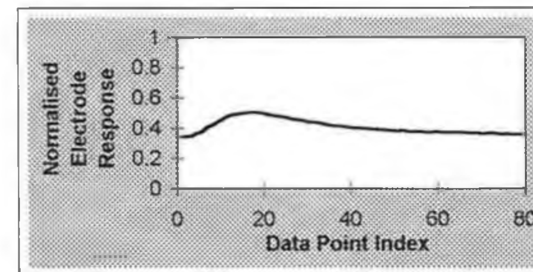
Pattern 7



Na ISE

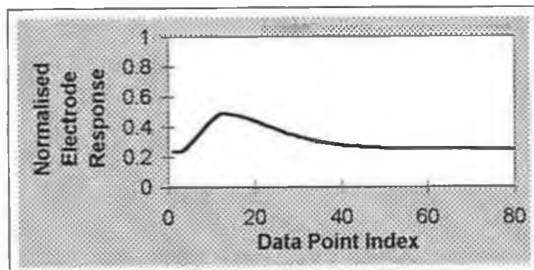


K ISE

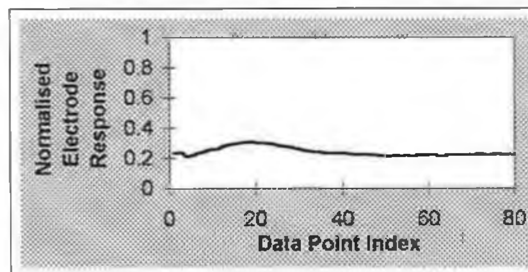


Ca ISE

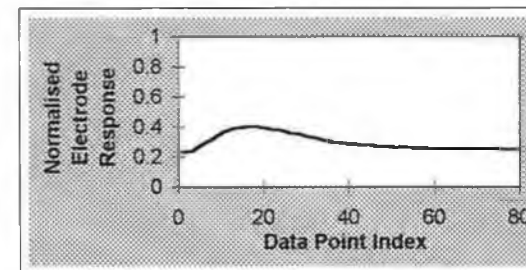
Pattern 8



Na ISE

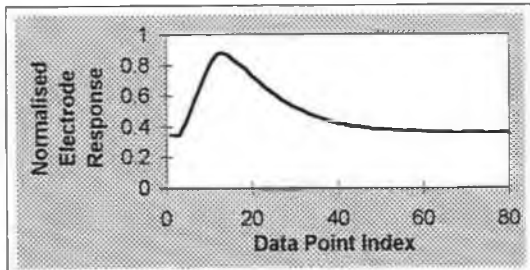


K ISE

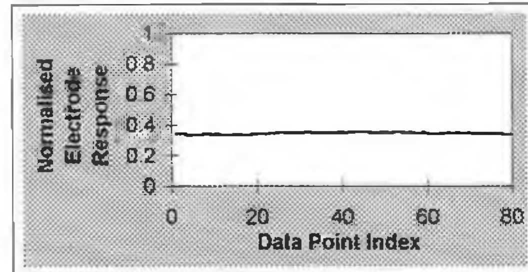


Ca ISE

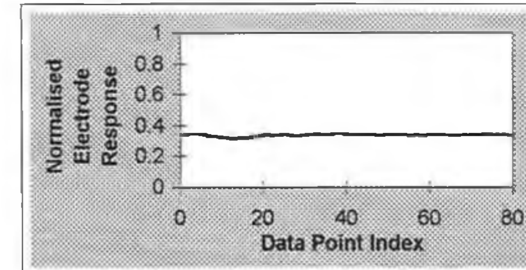
Pattern 9



Na ISE

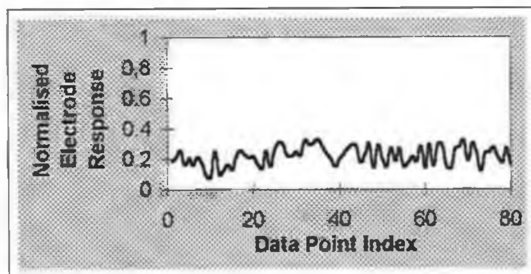


K ISE

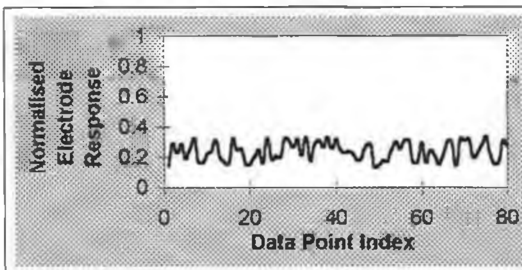


Ca ISE

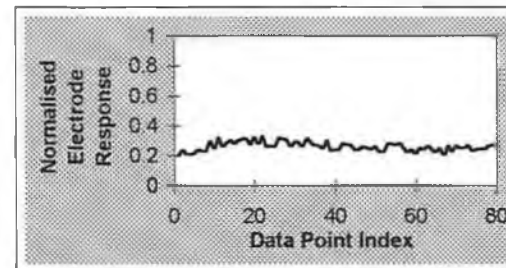
Pattern 10



Na ISE

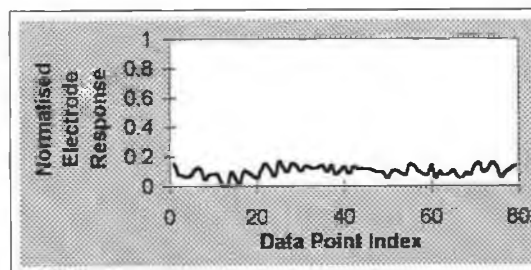


K ISE

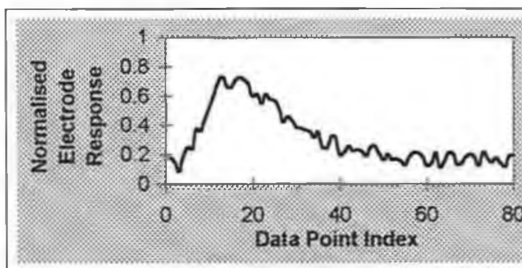


Ca ISE

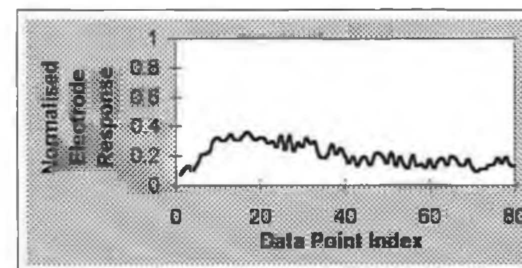
Pattern 11



Na ISE

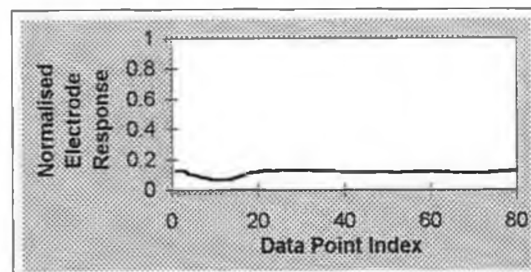


K ISE

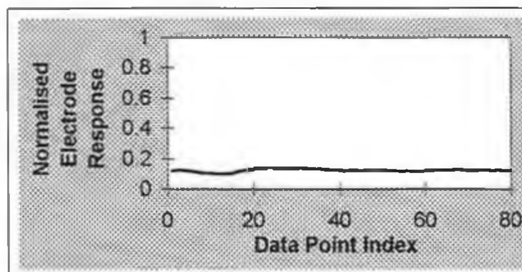


Ca ISE

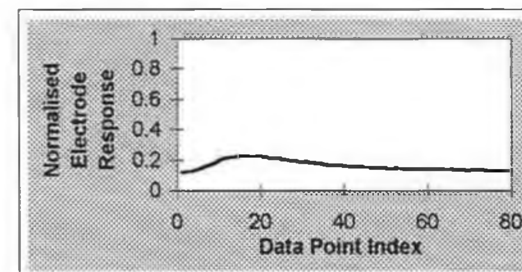
Pattern 12



Na ISE

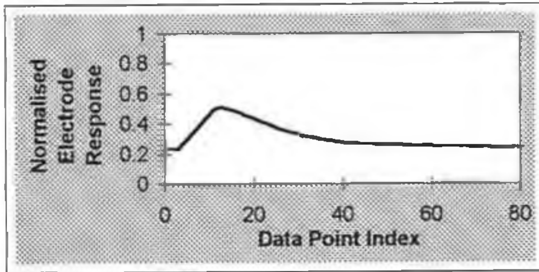


K ISE

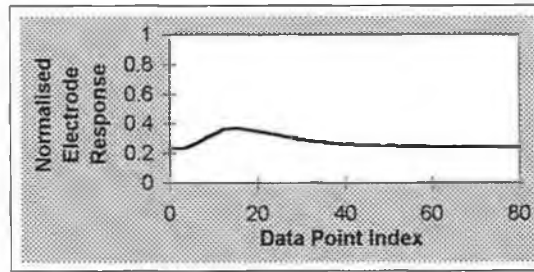


Ca ISE

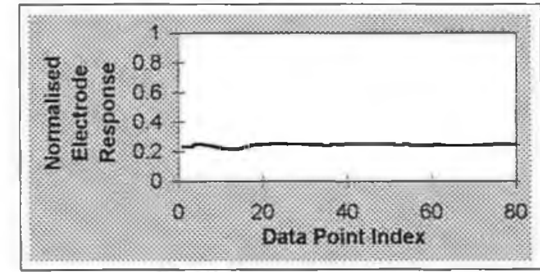
Pattern 13



Na ISE

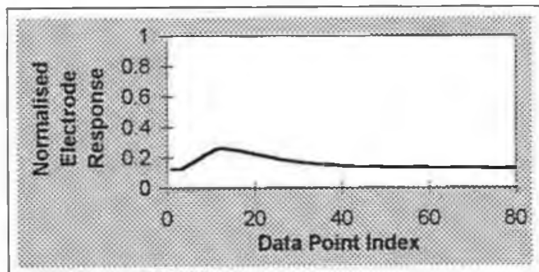


K ISE

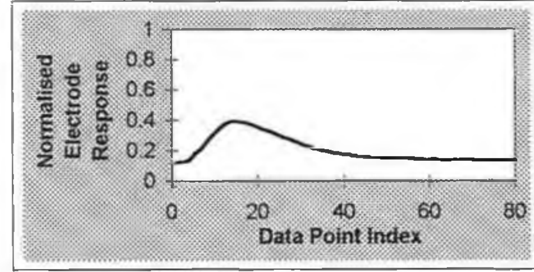


Ca ISE

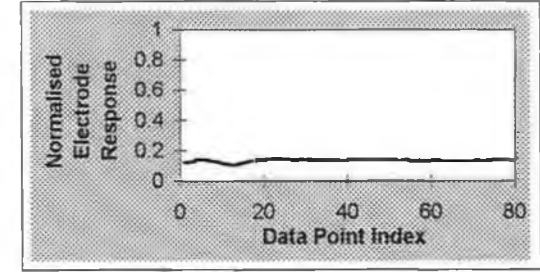
Pattern 14



Na ISE

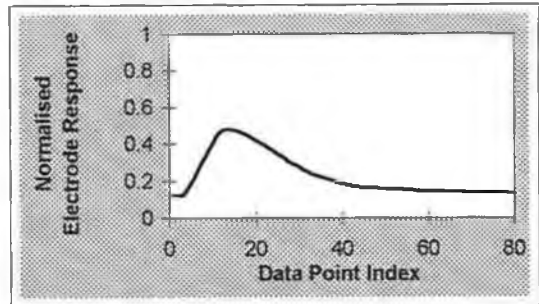


K ISE

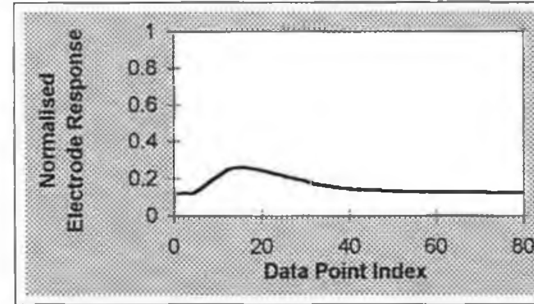


Ca ISE

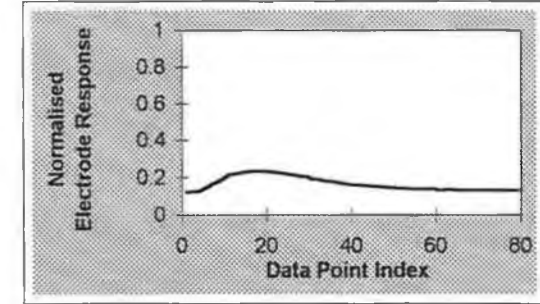
Pattern 15



Na ISE

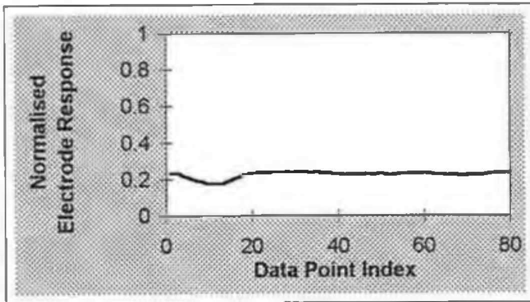


K ISE

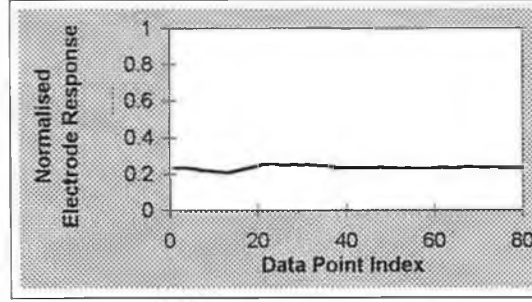


Ca ISE

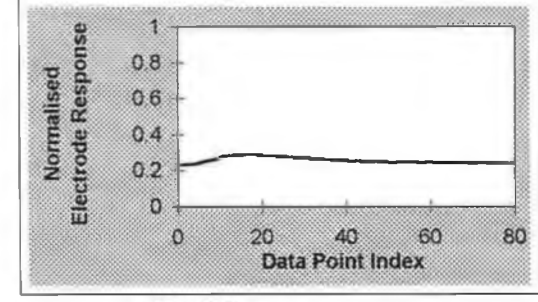
Pattern 16



Na ISE

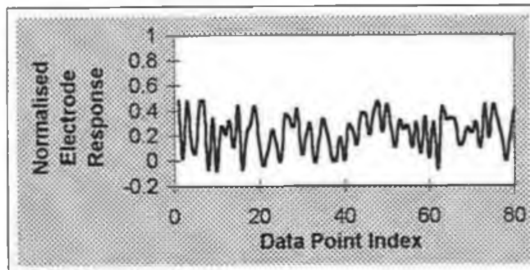


K ISE

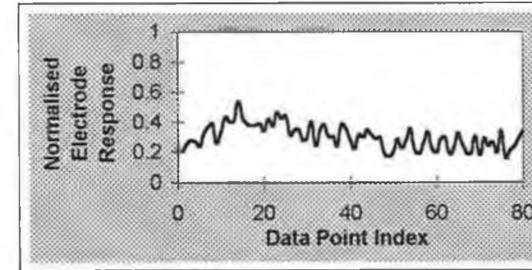


Ca ISE

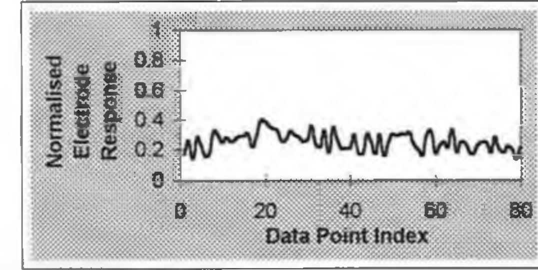
Pattern 17



Na ISE

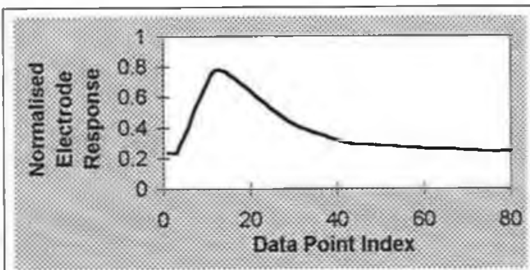


K ISE

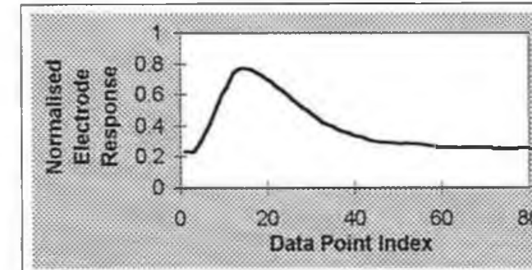


Ca ISE

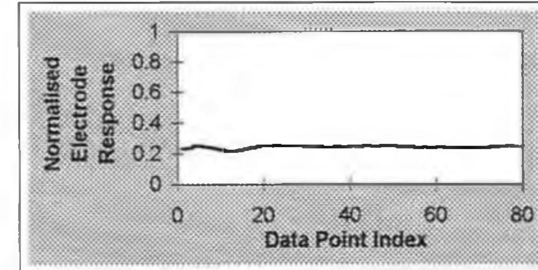
Pattern 18



Na ISE

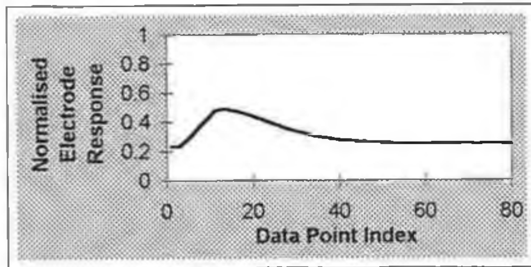


K ISE

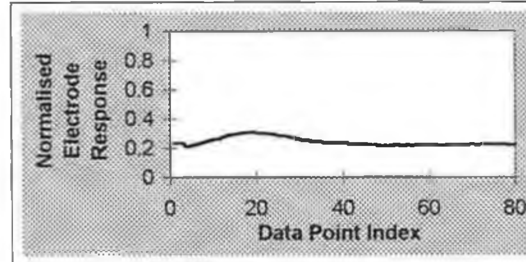


Ca ISE

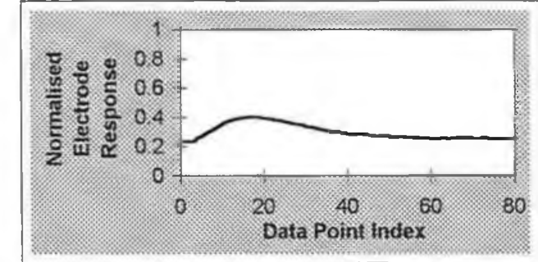
Pattern 19



Na ISE

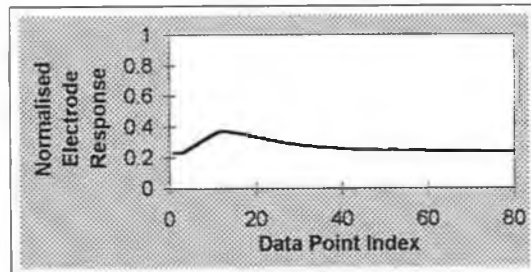


K ISE

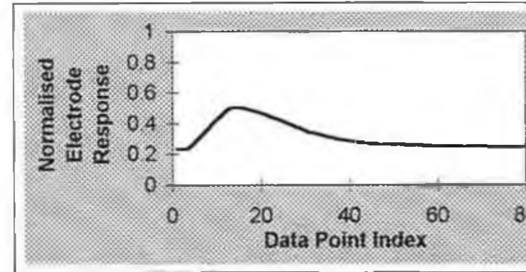


Ca ISE

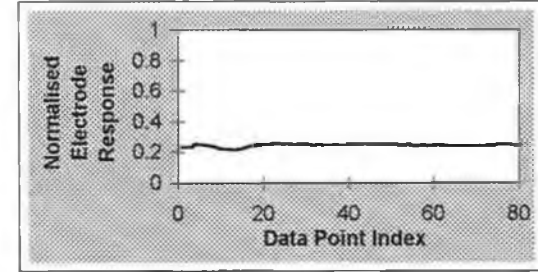
Pattern 20



Na ISE

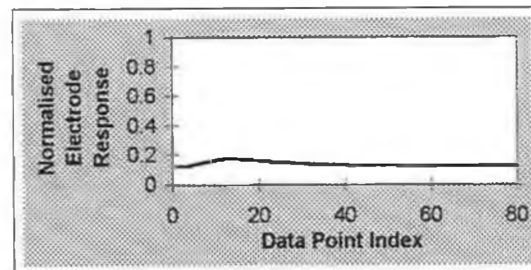


K ISE

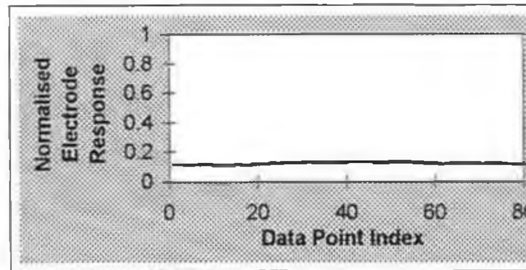


Ca ISE

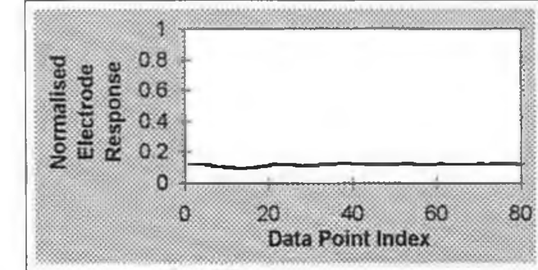
Pattern 21



Na ISE

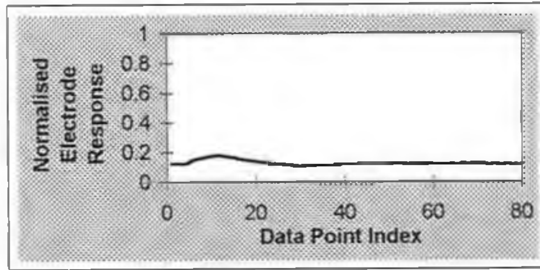


K ISE

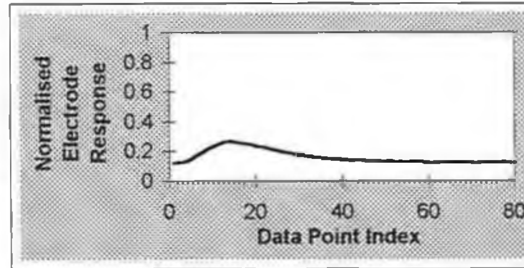


Ca ISE

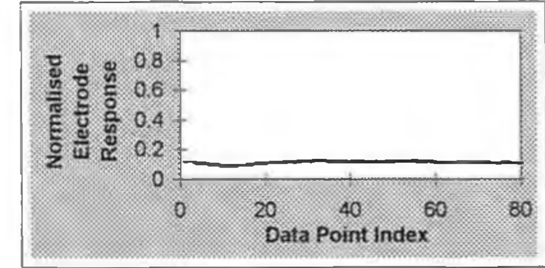
Pattern 22



Na ISE

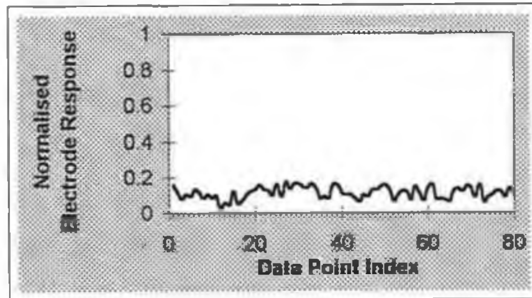


K ISE

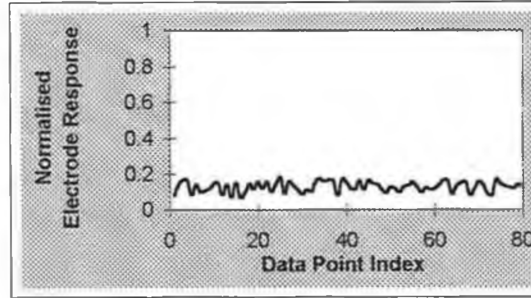


Ca ISE

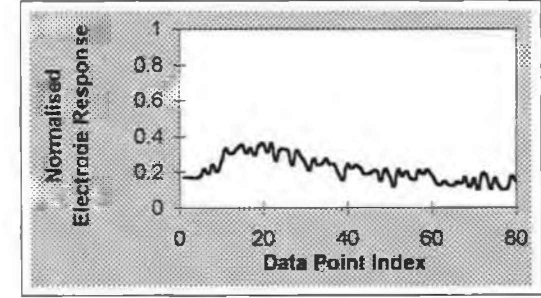
Pattern 23



Na ISE

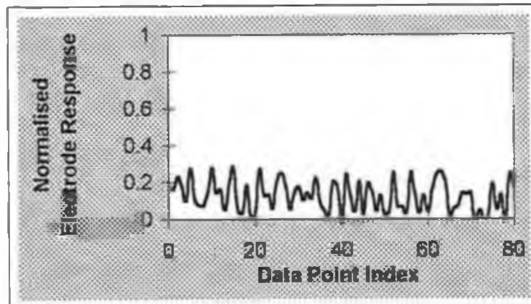


K ISE

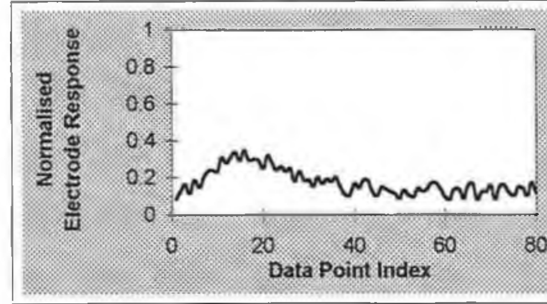


Ca ISE

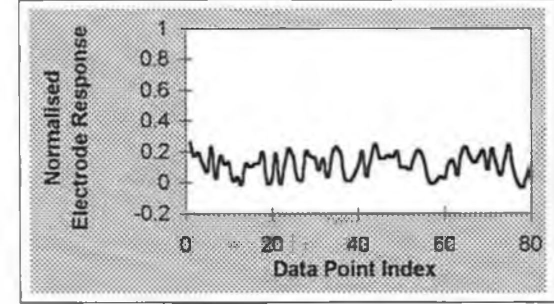
Pattern 24



Na ISE

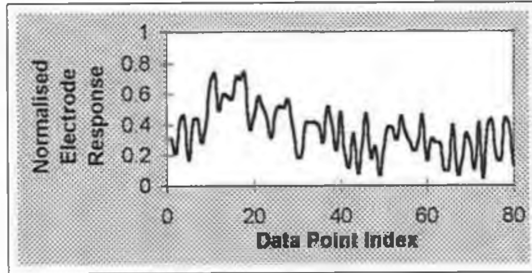


K ISE

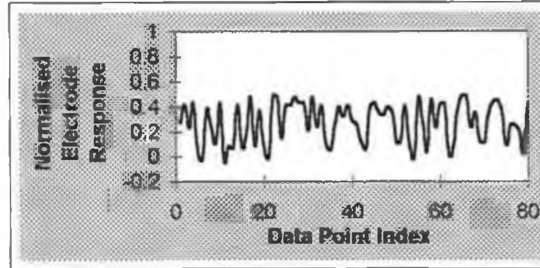


Ca ISE

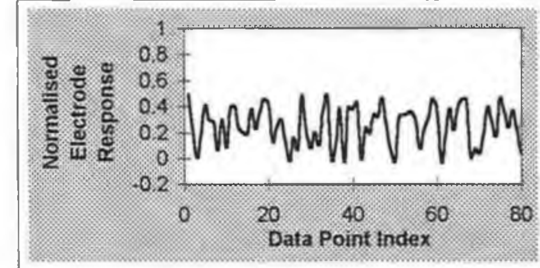
Pattern 25



Na ISE

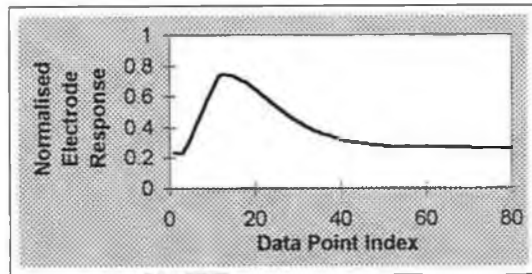


K ISE

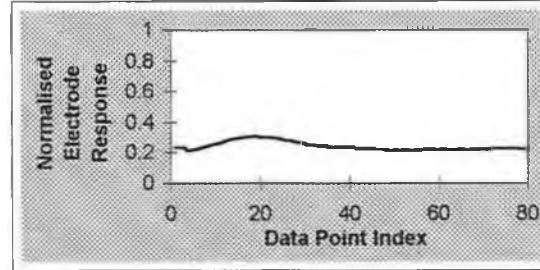


Ca ISE

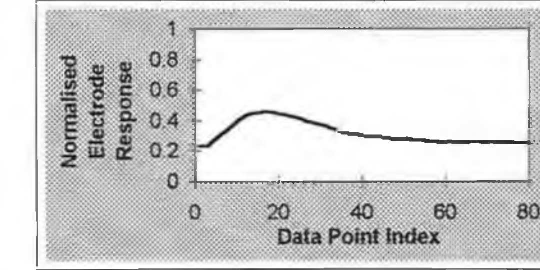
Pattern 26



Na ISE

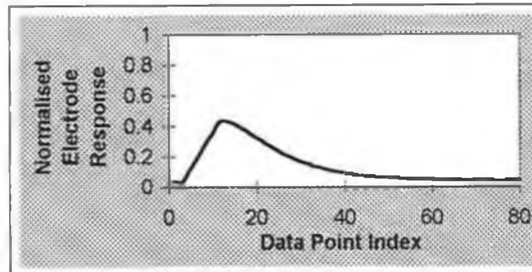


K ISE

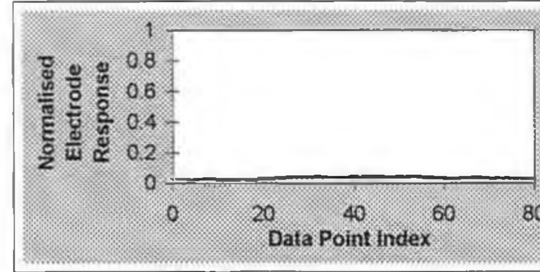


Ca ISE

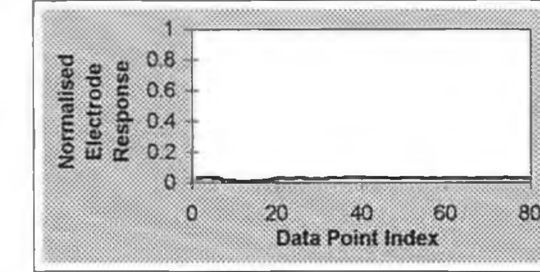
Pattern 27



Na ISE

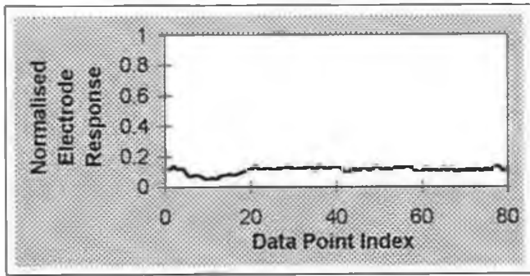


K ISE

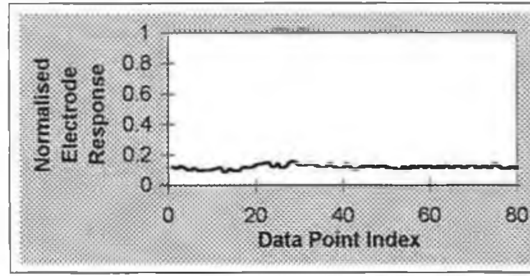


Ca ISE

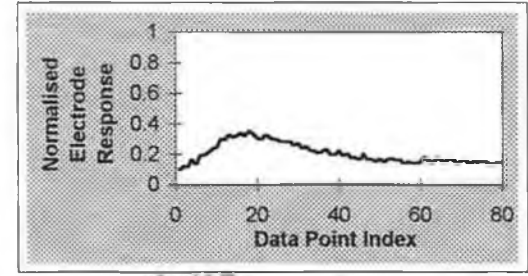
Pattern 28



Na ISE

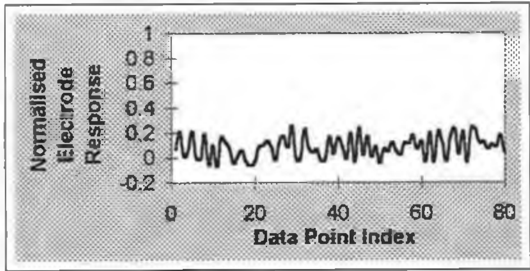


K ISE

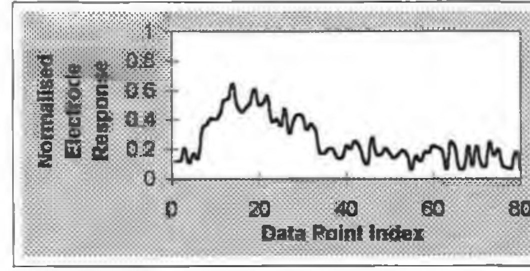


Ca ISE

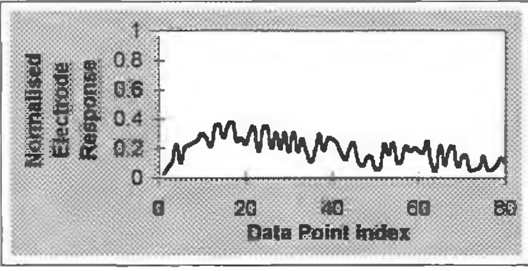
Pattern 29



Na ISE

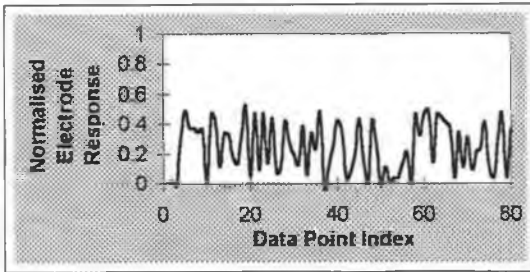


K ISE

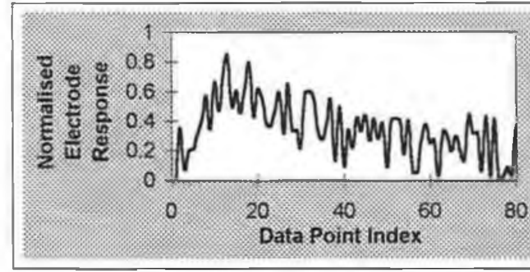


Ca ISE

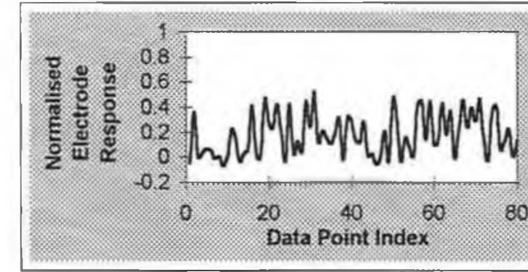
Pattern 30



Na ISE

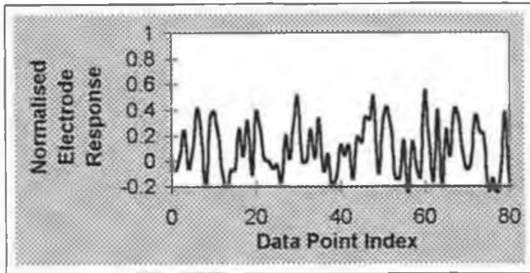


K ISE

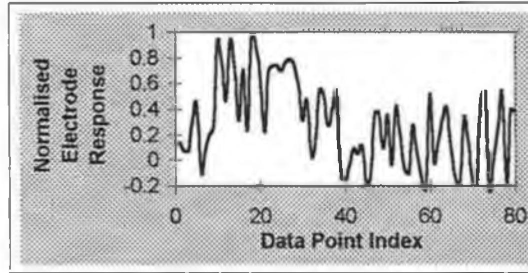


Ca ISE

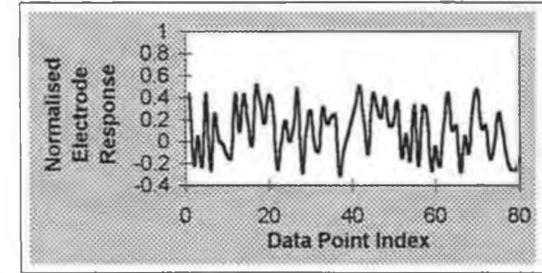
Pattern 31



Na ISE

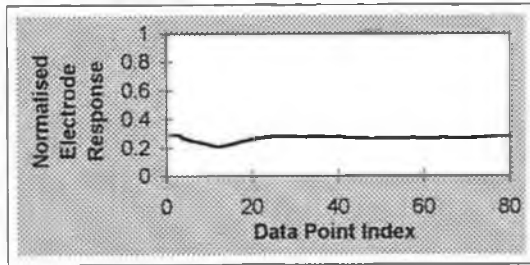


K ISE

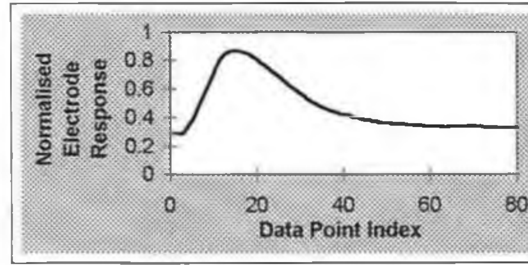


Ca ISE

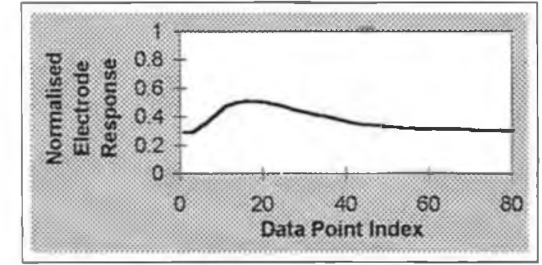
Pattern 32



Na ISE

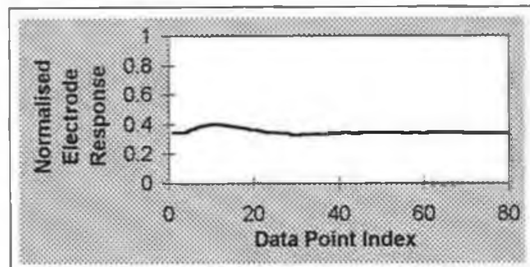


K ISE

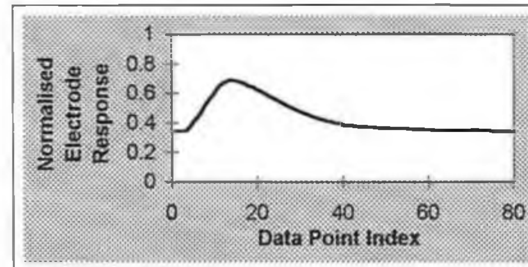


Ca ISE

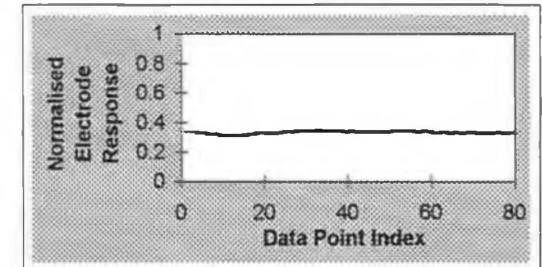
Pattern 33



Na ISE

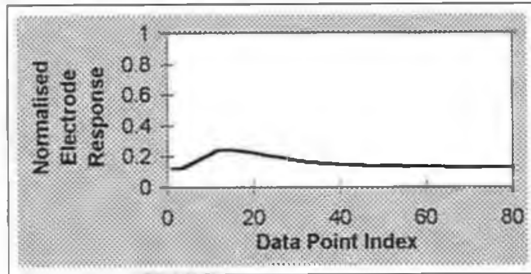


K ISE

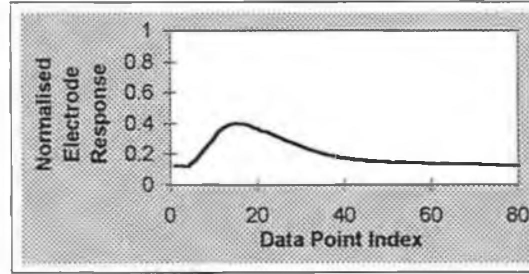


Ca ISE

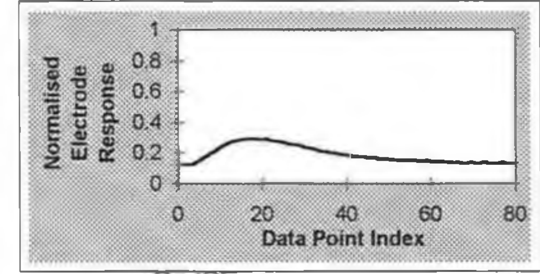
Pattern 34



Na ISE

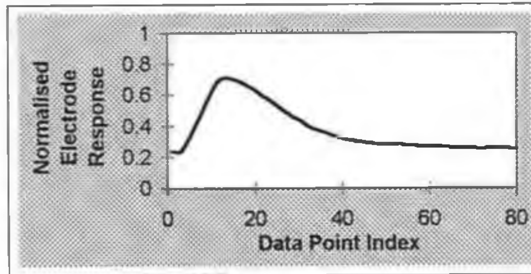


K ISE

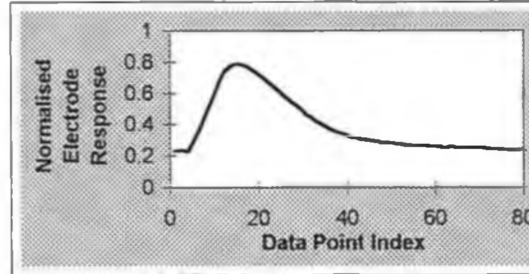


Ca ISE

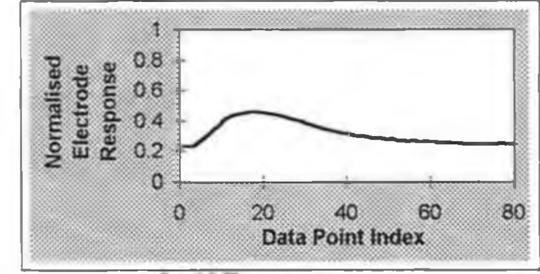
Pattern 35



Na ISE

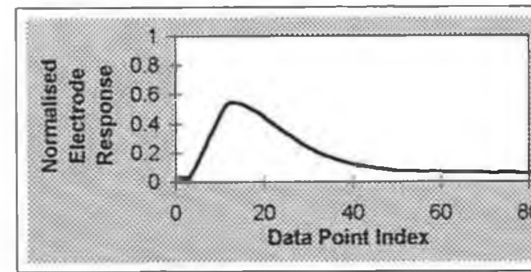


K ISE

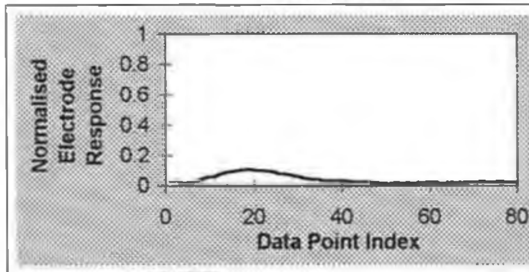


Ca ISE

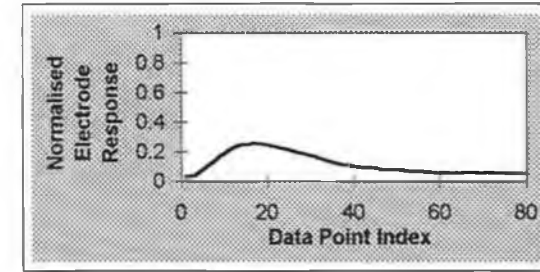
Pattern 36



Na ISE

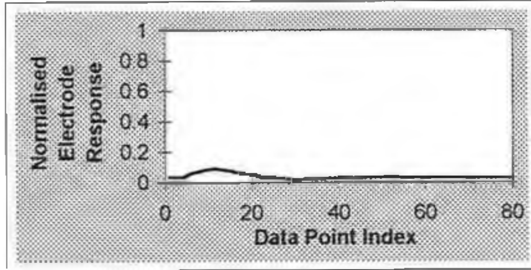


K ISE

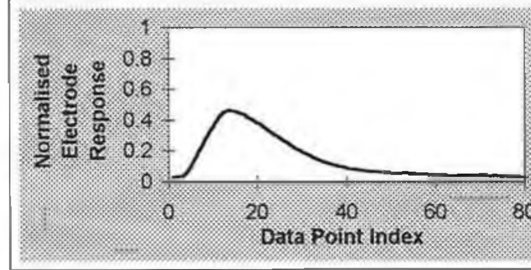


Ca ISE

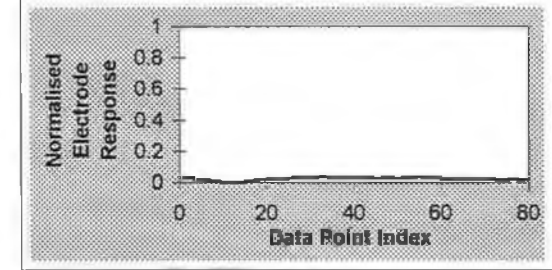
Pattern 37



Na ISE

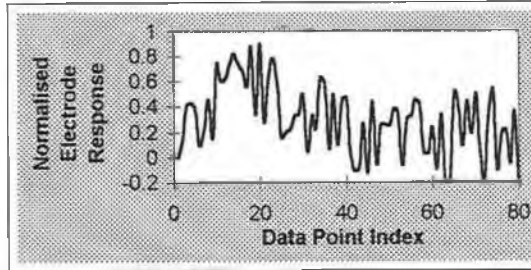


K ISE

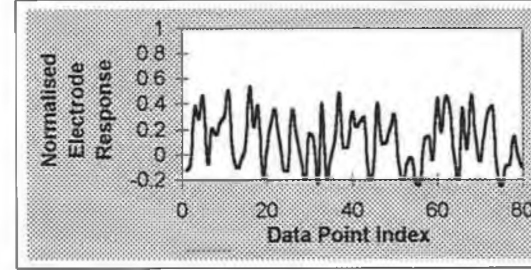


Ca ISE

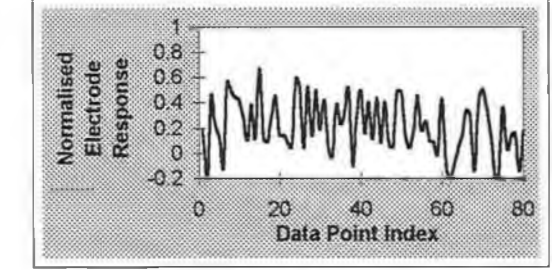
Pattern 38



Na ISE

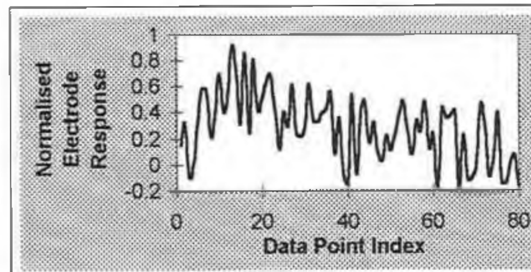


K ISE

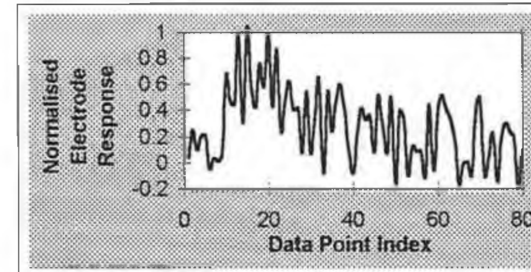


Ca ISE

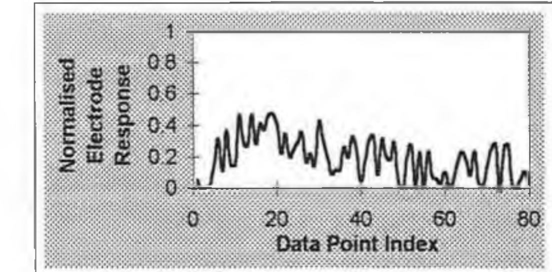
Pattern 39



Na ISE

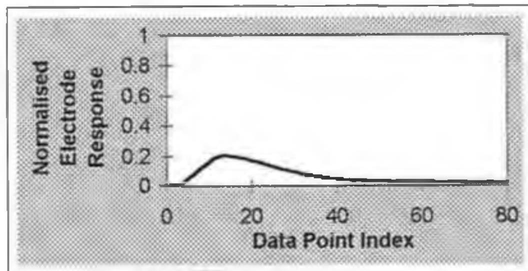


K ISE

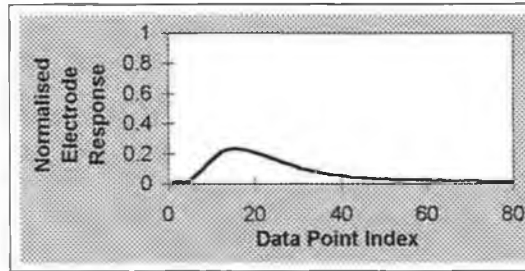


Ca ISE

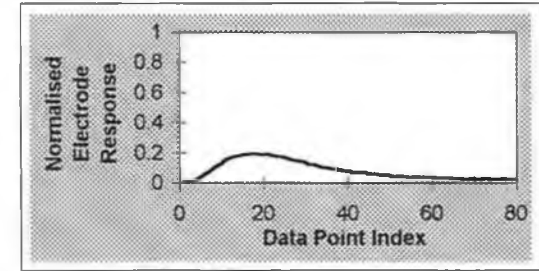
Pattern 40



Na ISE

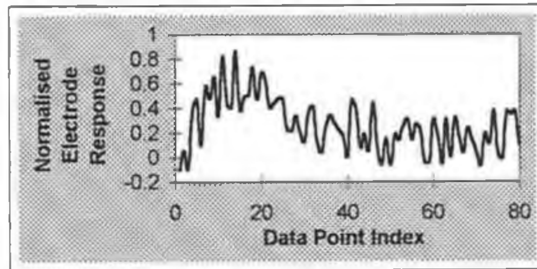


K ISE

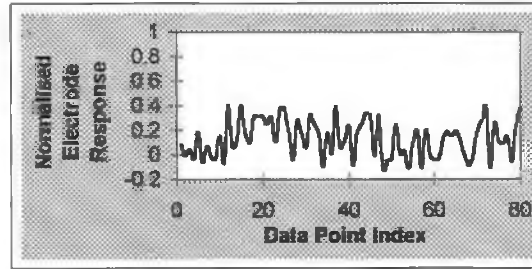


Ca ISE

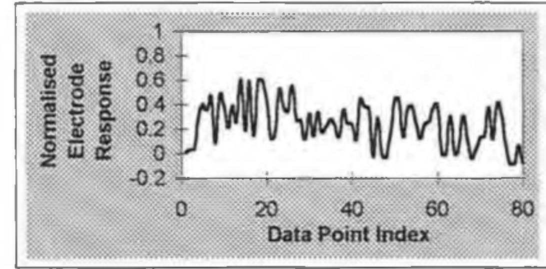
Pattern 41



Na ISE

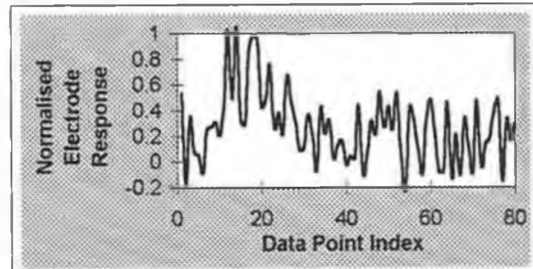


K ISE

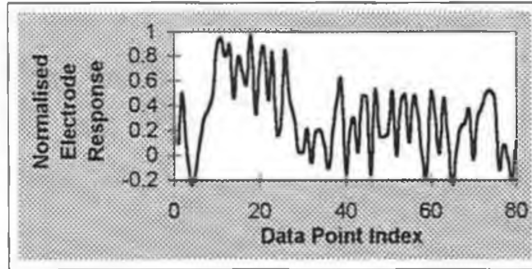


Ca ISE

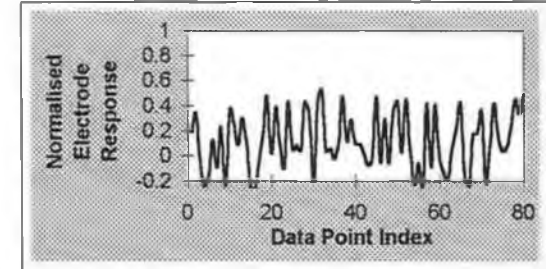
Pattern 42



Na ISE

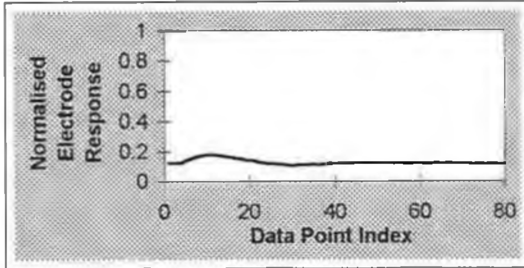


K ISE

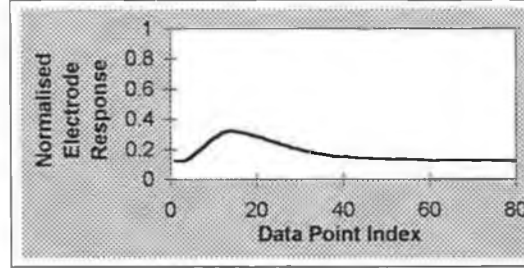


Ca ISE

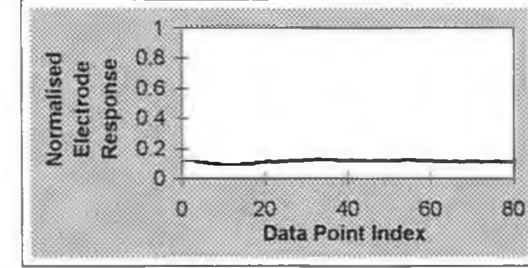
Pattern 43



Na ISE

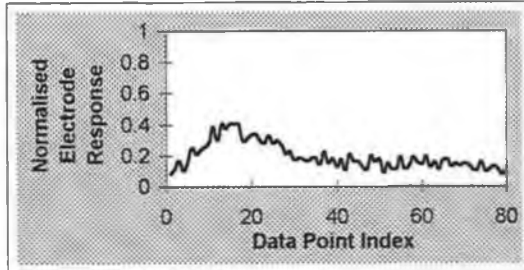


K ISE

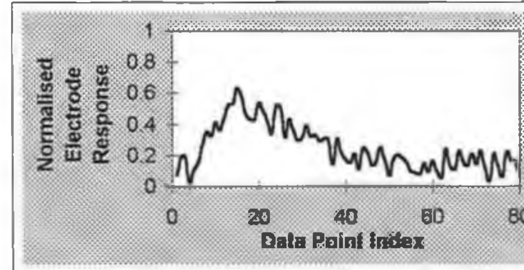


Ca ISE

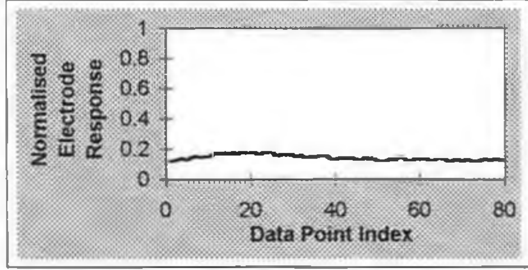
Pattern 44



Na ISE

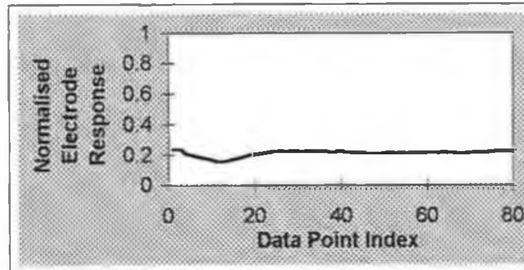


K ISE

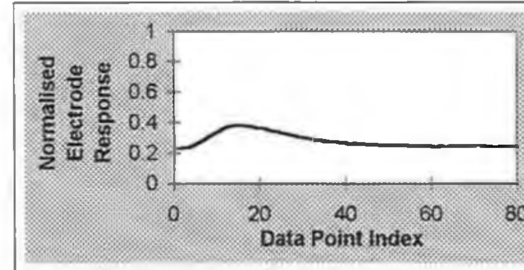


Ca ISE

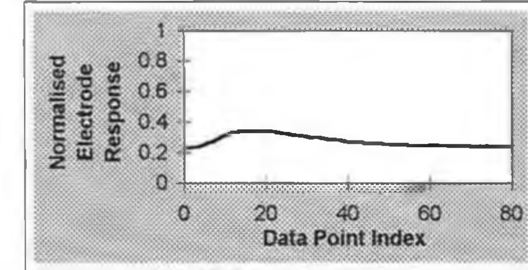
Pattern 45



Na ISE

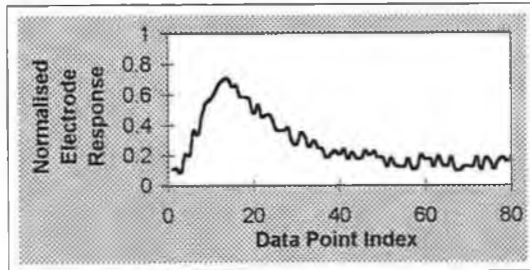


K ISE

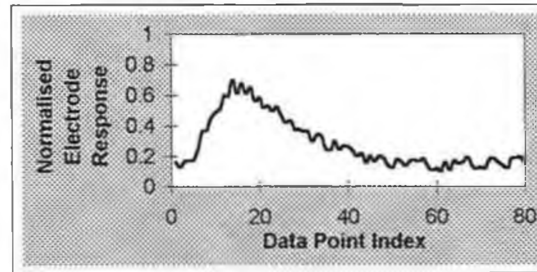


Ca ISE

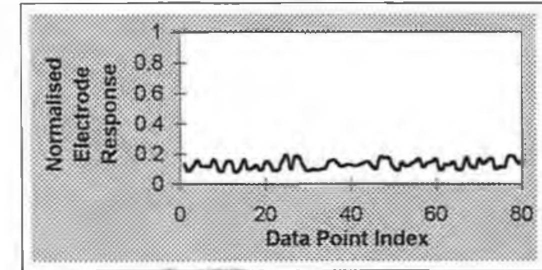
Pattern 46



Na ISE

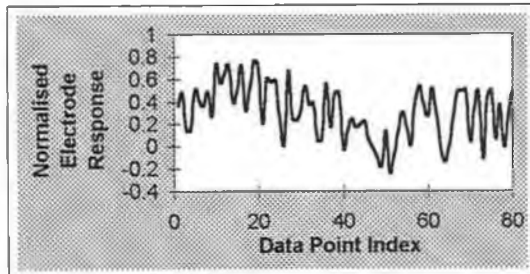


K ISE

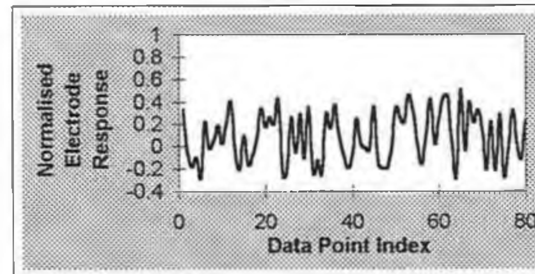


Ca ISE

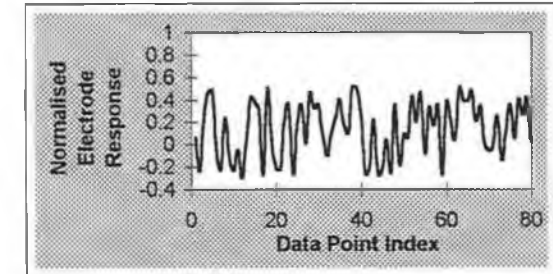
Pattern 47



Na ISE

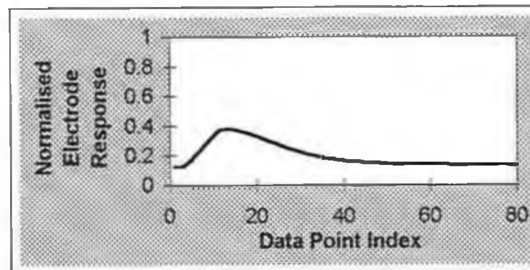


K ISE

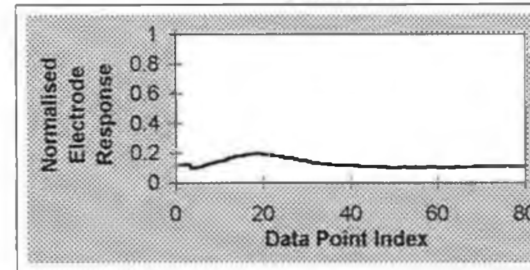


Ca ISE

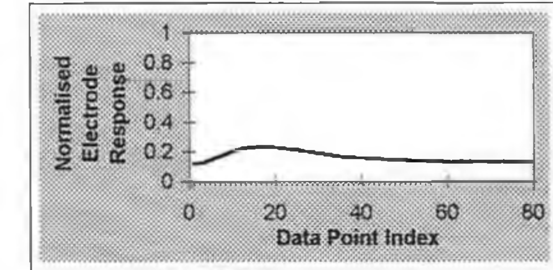
Pattern 48



Na ISE

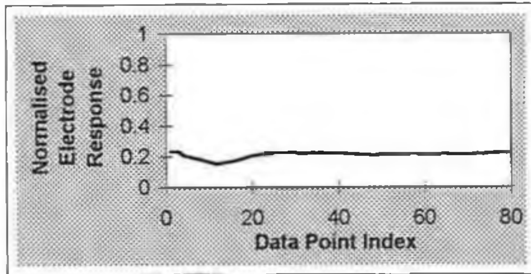


K ISE

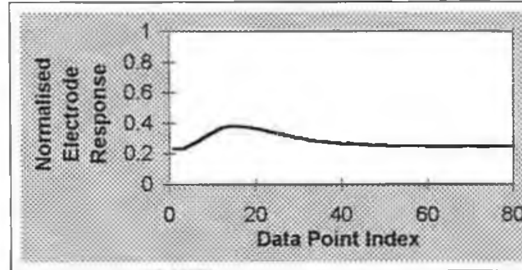


Ca ISE

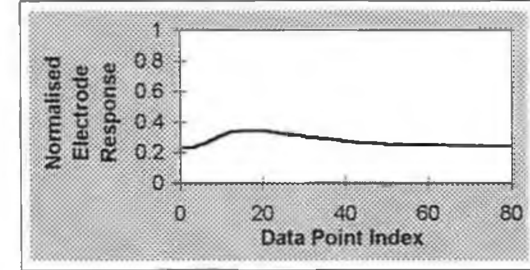
Pattern 49



Na ISE

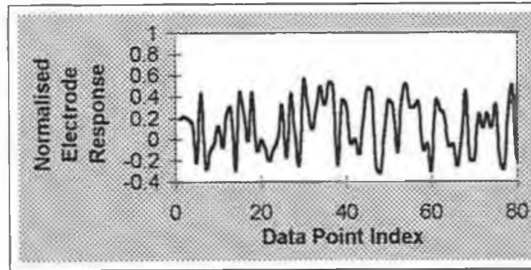


K ISE

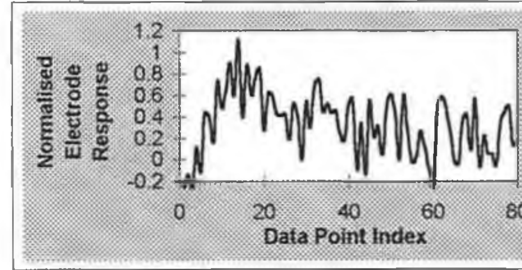


Ca ISE

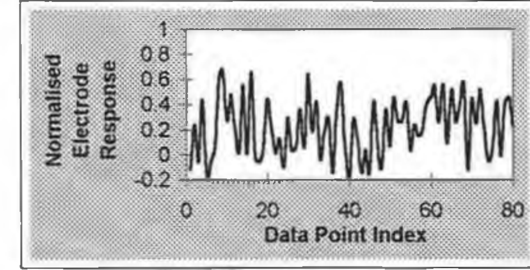
Pattern 50



Na ISE

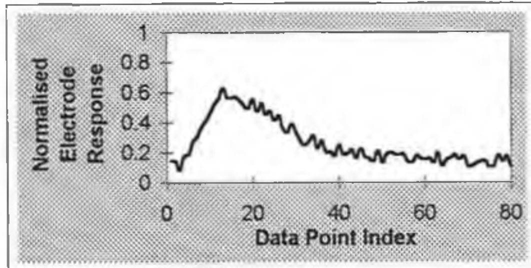


K ISE

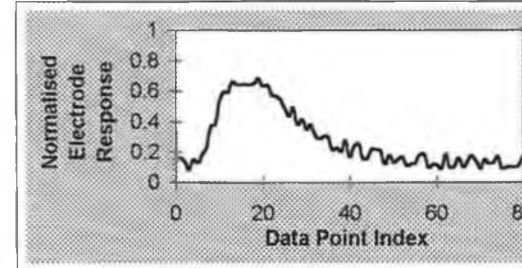


Ca ISE

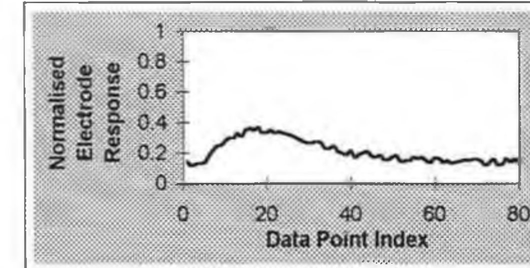
Pattern 51



Na ISE

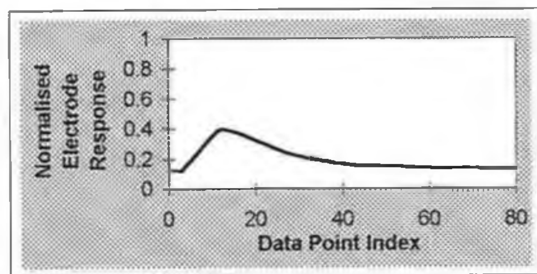


K ISE

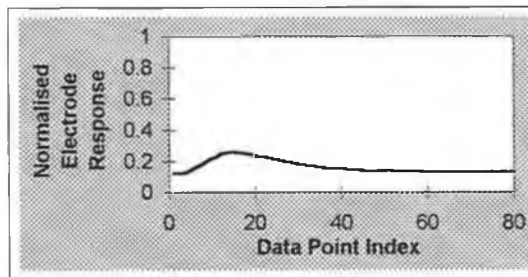


Ca ISE

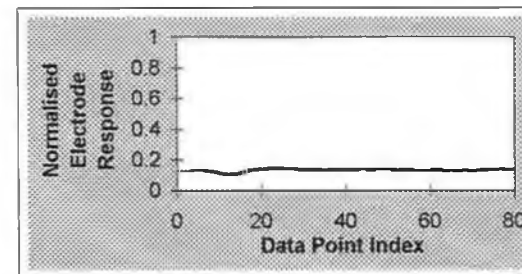
Pattern 52



Na ISE

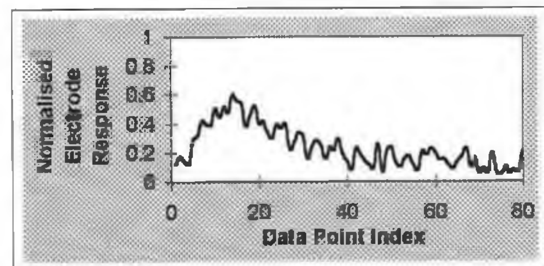


K ISE

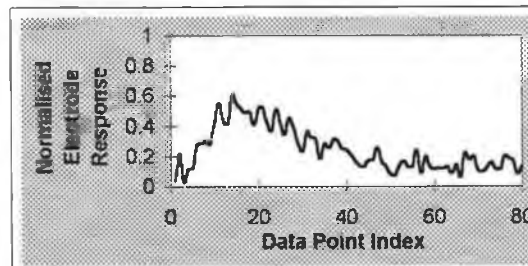


Ca ISE

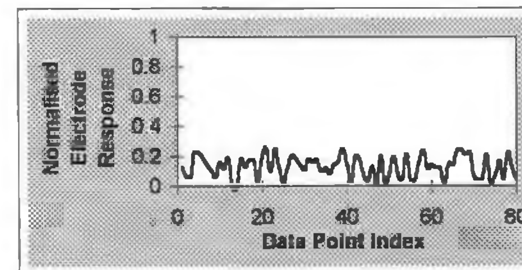
Pattern 53



Na ISE

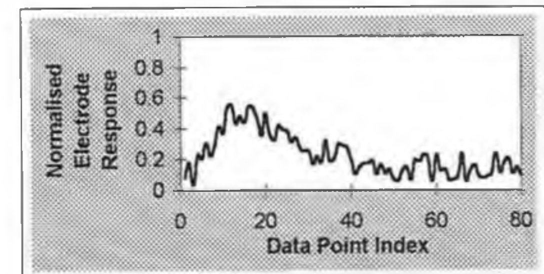


K ISE

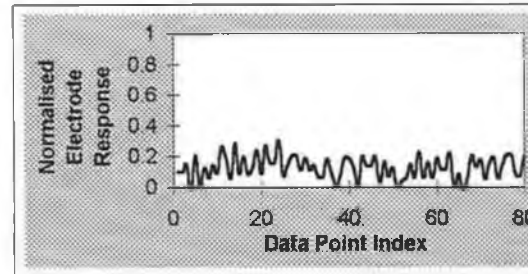


Ca ISE

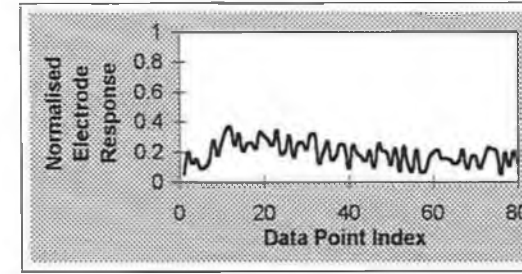
Pattern 54



Na ISE

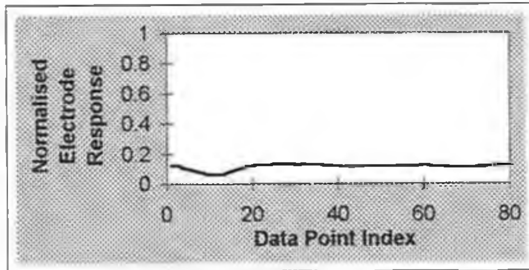


K ISE

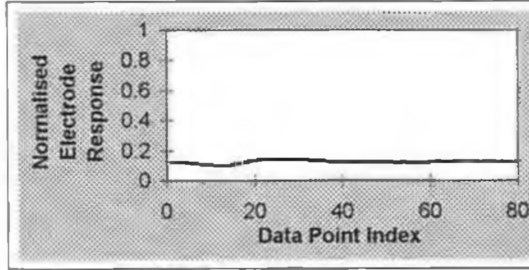


Ca ISE

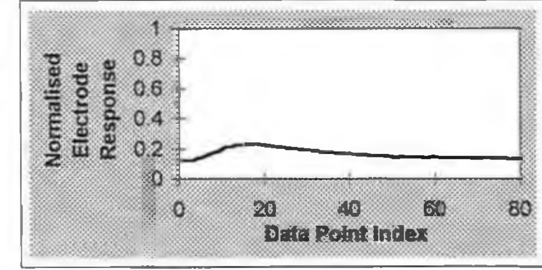
Pattern 55



Na ISE

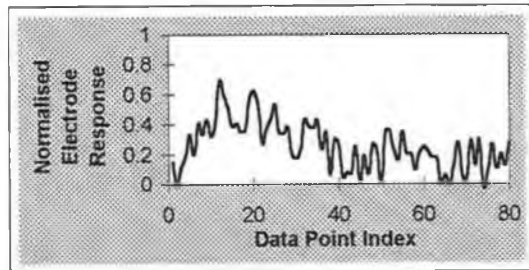


K ISE

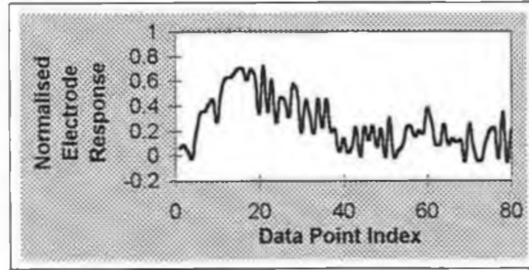


Ca ISE

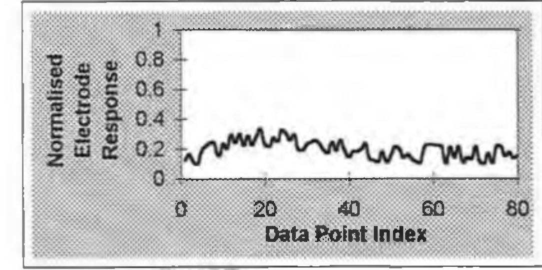
Pattern 56



Na ISE

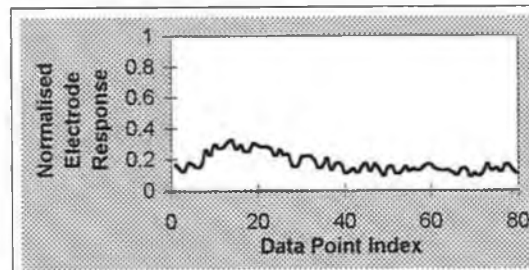


K ISE

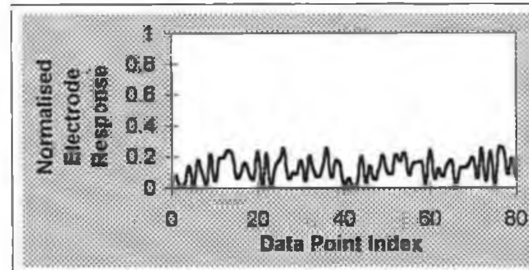


Ca ISE

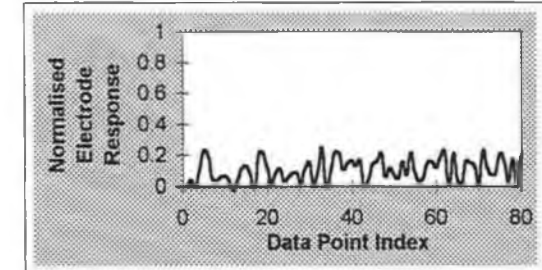
Pattern 57



Na ISE

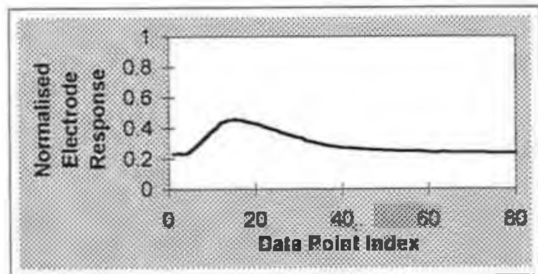


K ISE

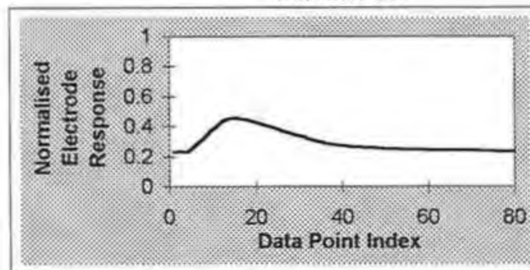


Ca ISE

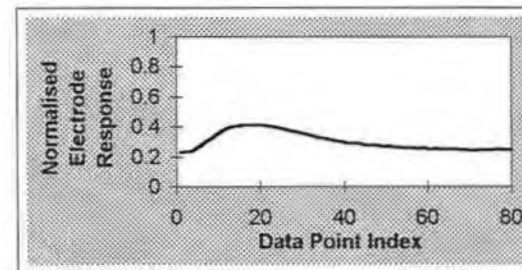
Pattern 58



Na ISE

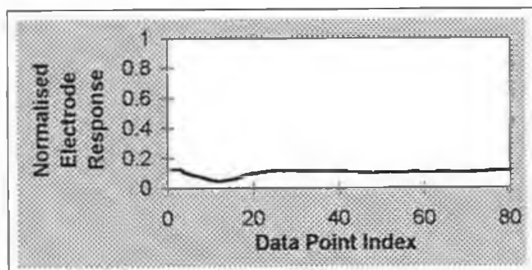


K ISE

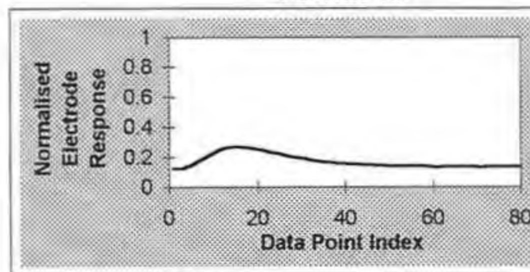


Ca ISE

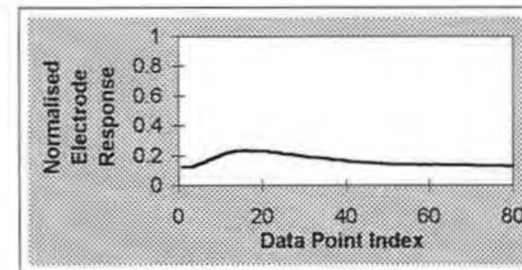
Pattern 59



Na ISE

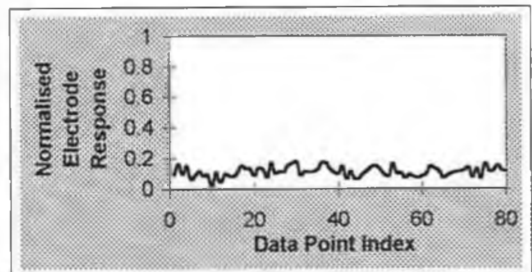


K ISE

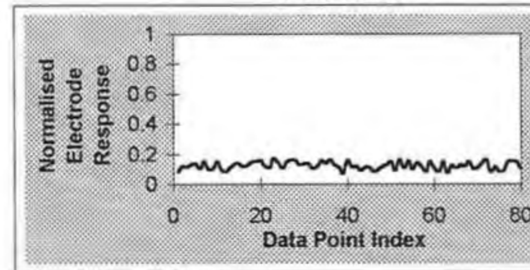


Ca ISE

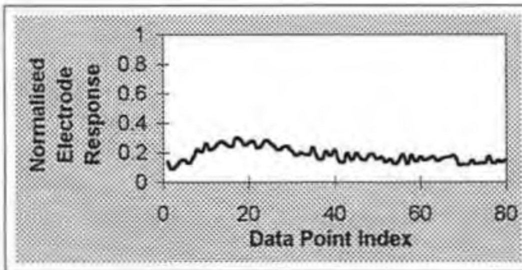
Pattern 60



Na ISE

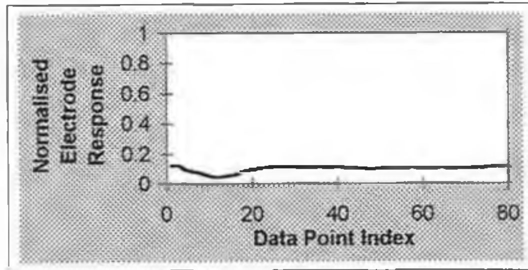


K ISE

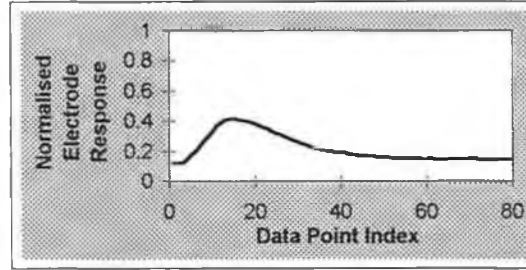


Ca ISE

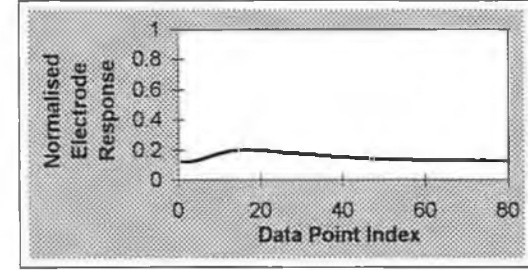
Pattern 61



Na ISE

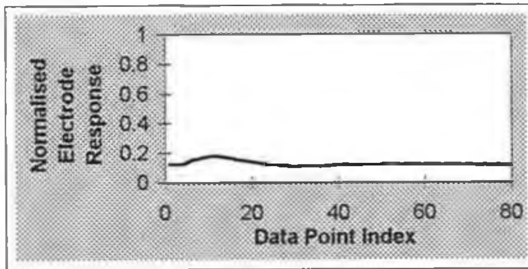


K ISE

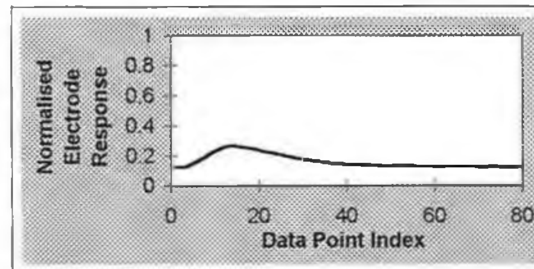


Ca ISE

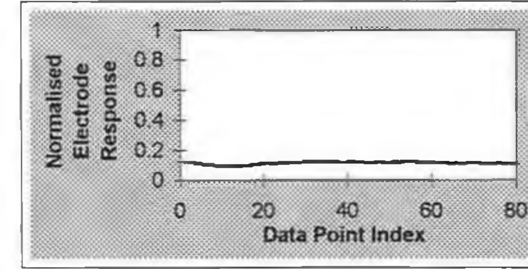
Pattern 62



Na ISE

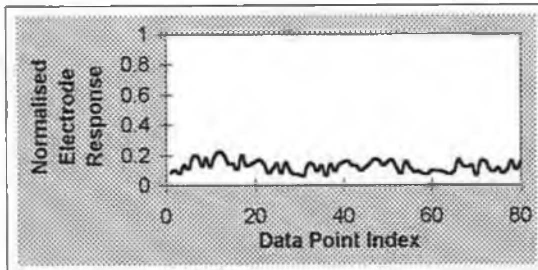


K ISE

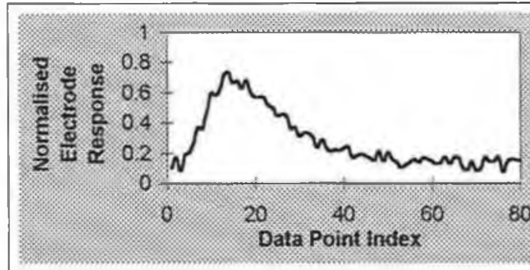


Ca ISE

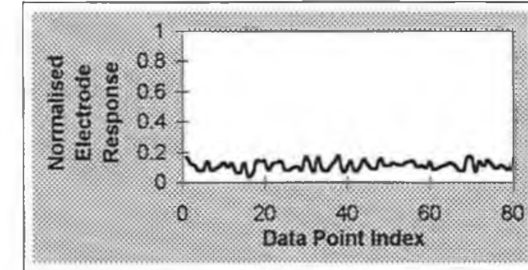
Pattern 63



Na ISE

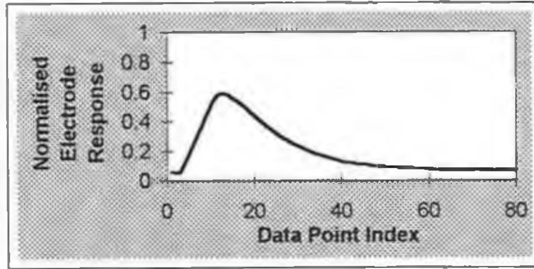


K ISE

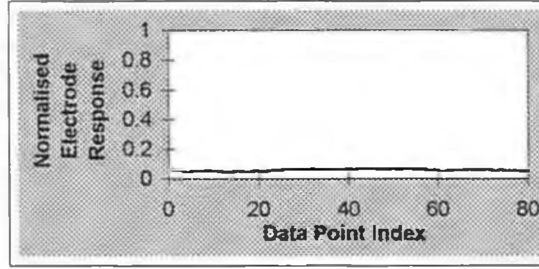


Ca ISE

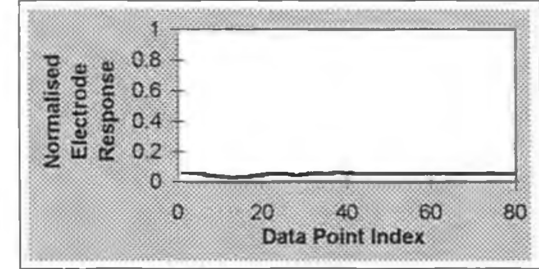
Pattern 64



Na ISE

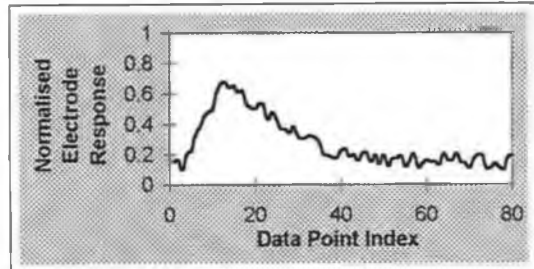


K ISE

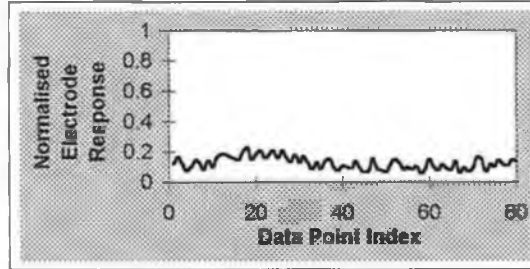


Ca ISE

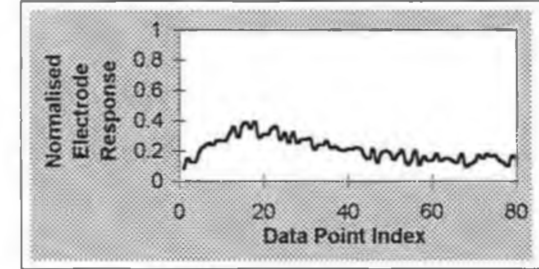
Pattern 65



Na ISE

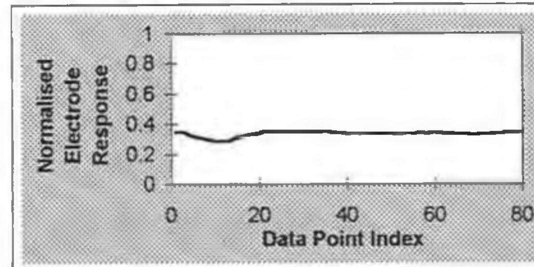


K ISE

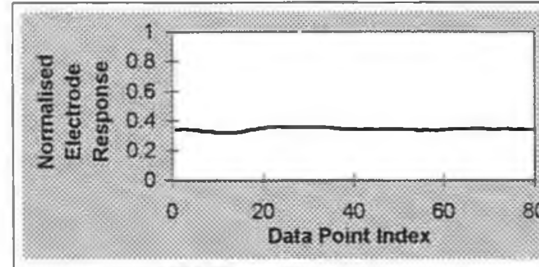


Ca ISE

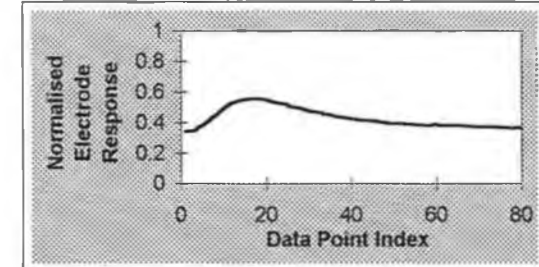
Pattern 66



Na ISE

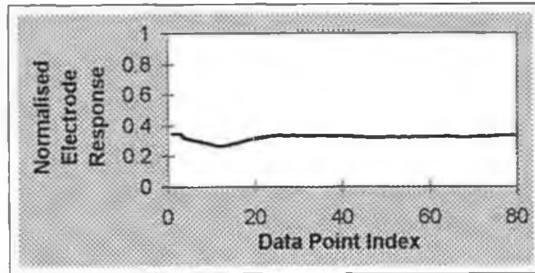


K ISE

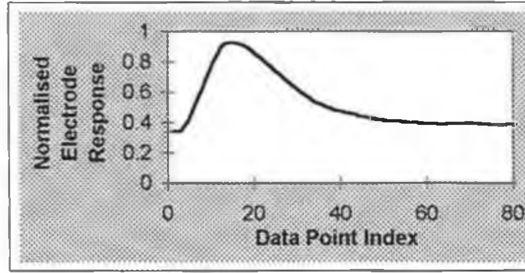


Ca ISE

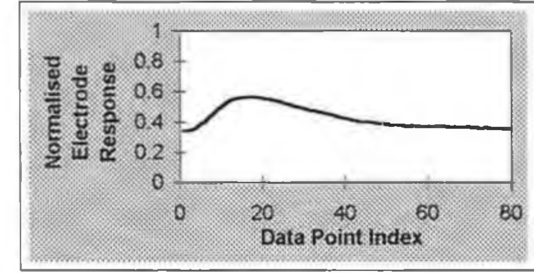
Pattern 67



Na ISE

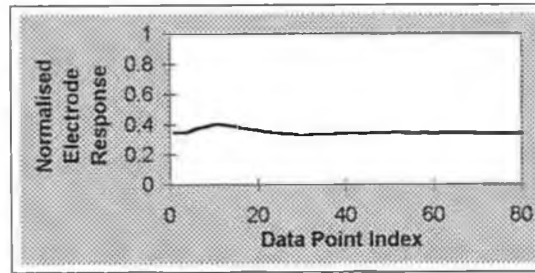


K ISE

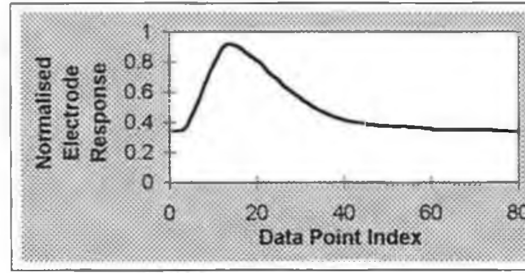


Ca ISE

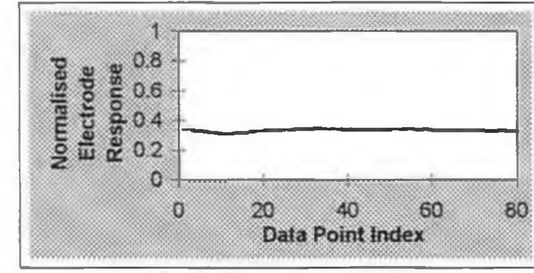
Pattern 68



Na ISE

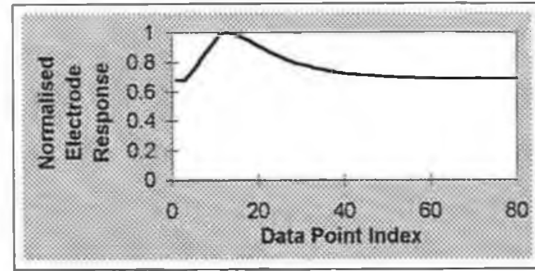


K ISE

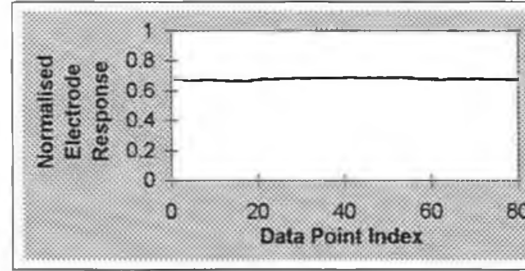


Ca ISE

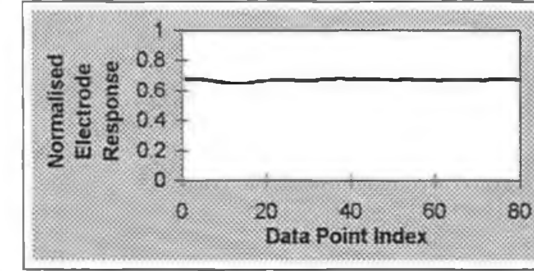
Pattern 69



Na ISE

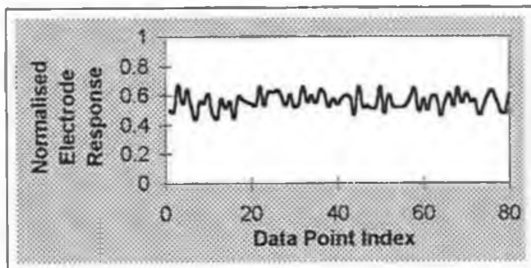


K ISE

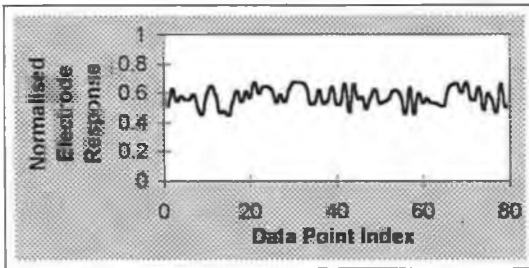


Ca ISE

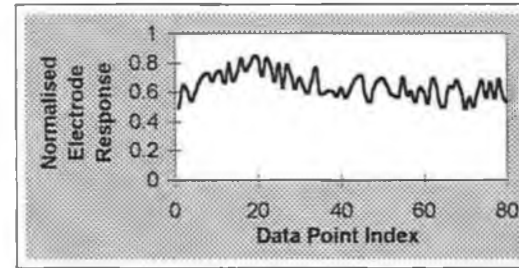
Pattern 70



Na ISE



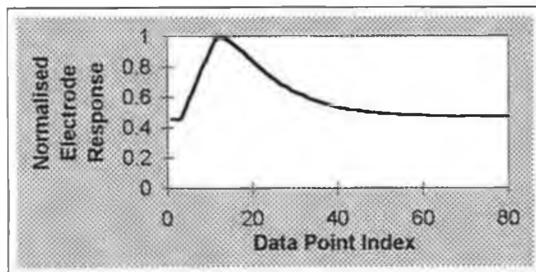
K ISE



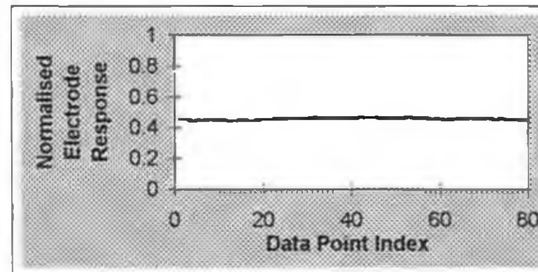
Ca ISE

Appendix 6 Patterns Used For Testing The Neural Network Described In Chapter 2 (See 2.3.2) (distortions described in tables 2.7 to 2.13)

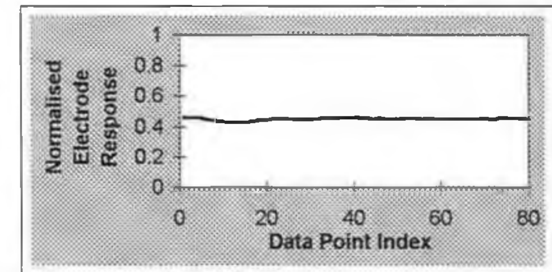
Pattern 1



Na ISE

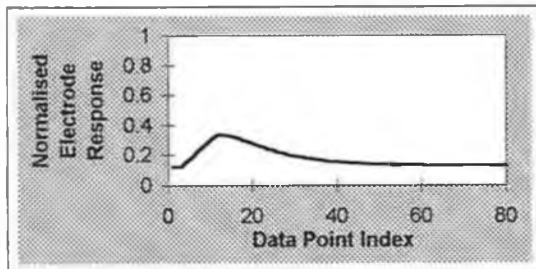


K ISE

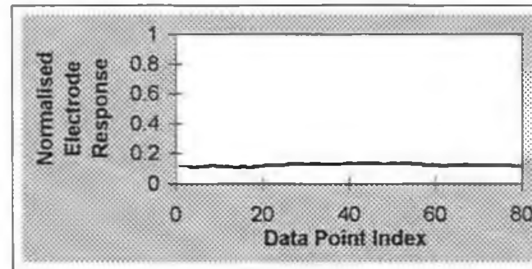


Ca ISE

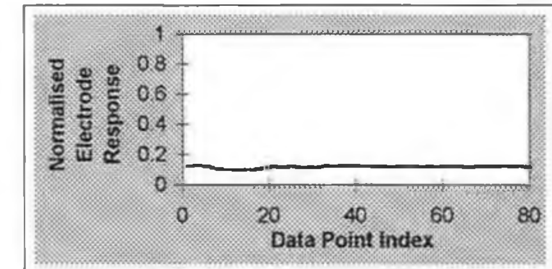
Pattern 2



Na ISE

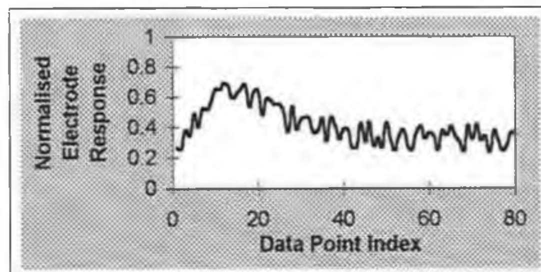


K ISE

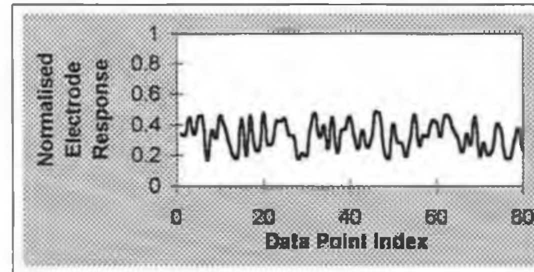


Ca ISE

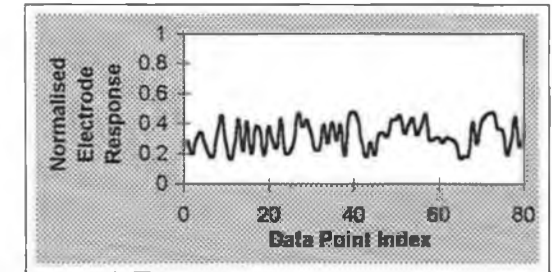
Pattern 3



Na ISE

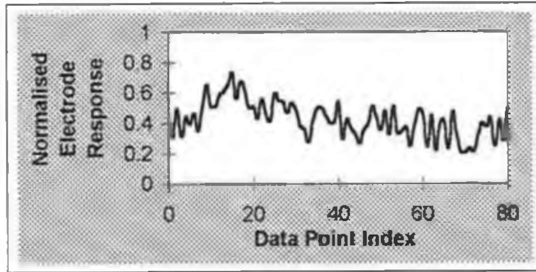


K ISE

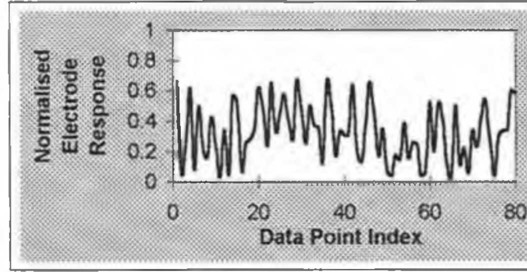


Ca ISE

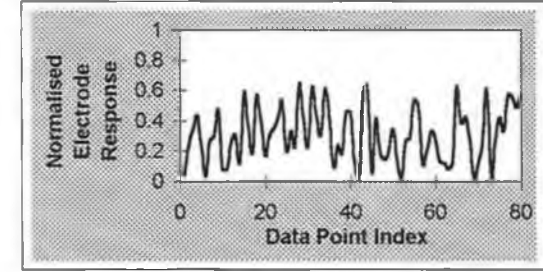
Pattern 4



Na ISE

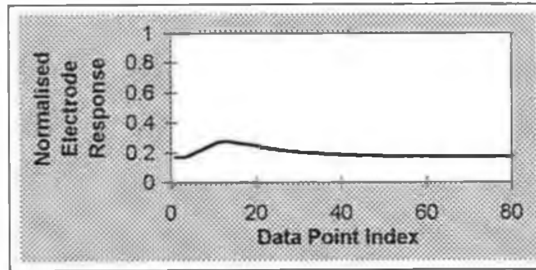


K ISE

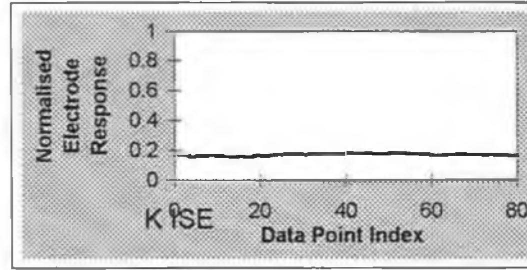


Ca ISE

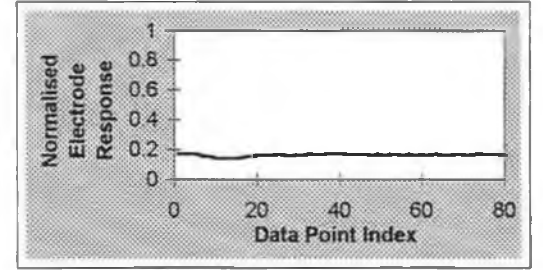
Pattern 5



Na ISE

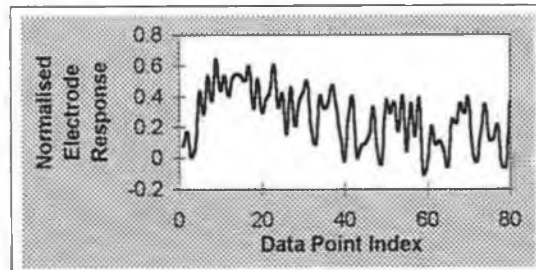


K ISE

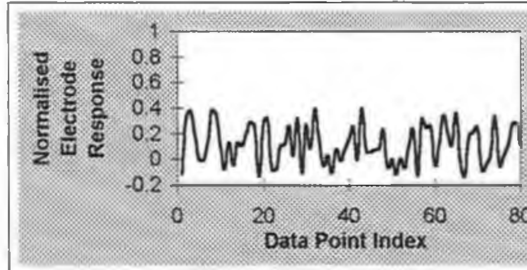


Ca ISE

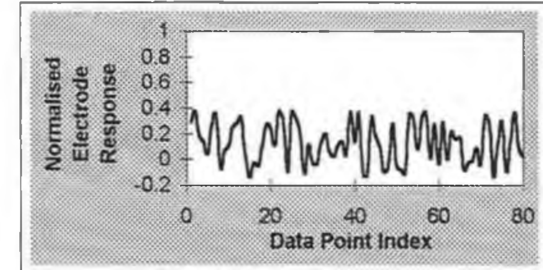
Pattern 6



Na ISE

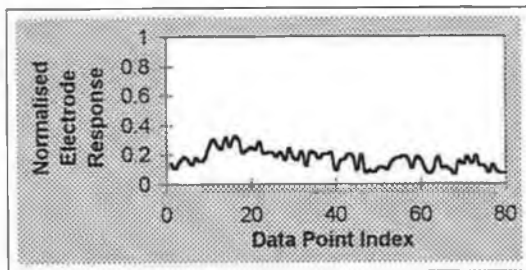


K ISE

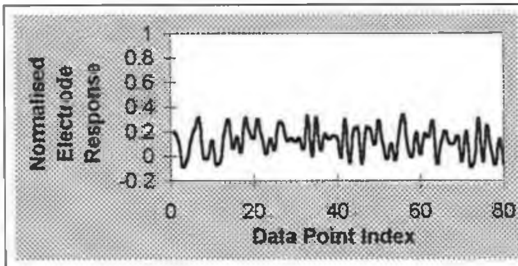


Ca ISE

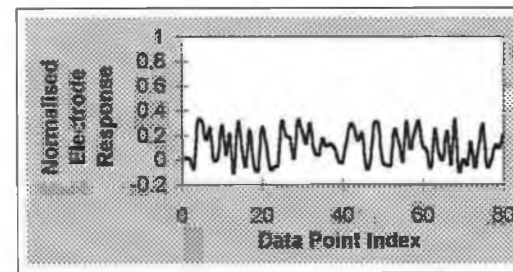
Pattern 7



Na ISE

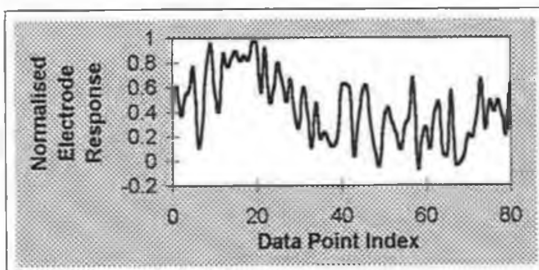


K ISE

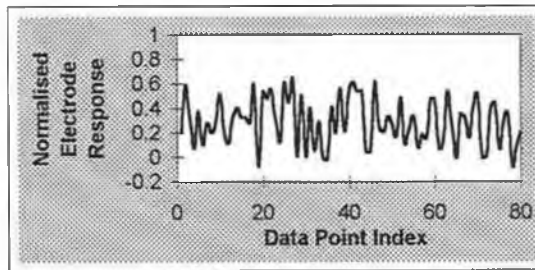


Ca ISE

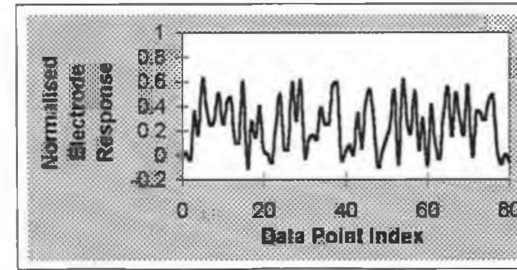
Pattern 8



Na ISE

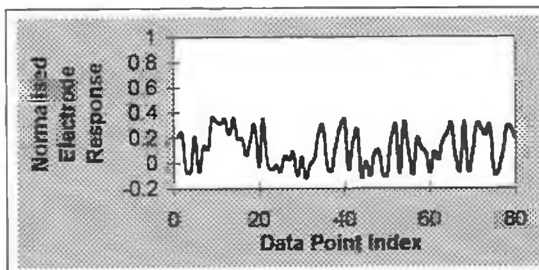


K ISE

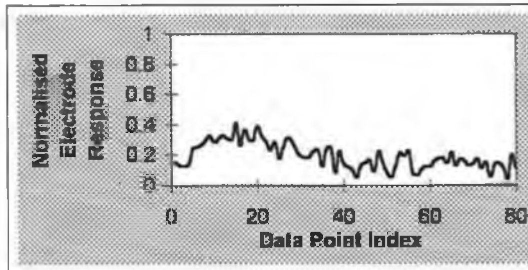


Ca ISE

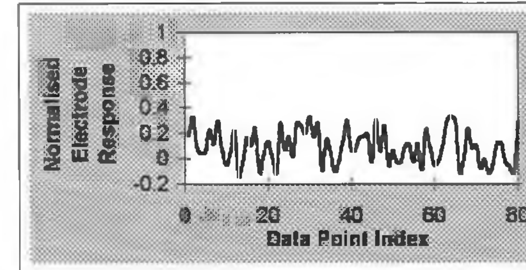
Pattern 9



Na ISE

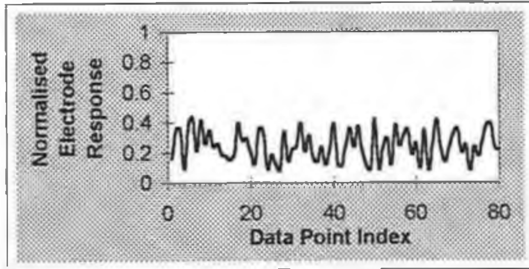


K ISE

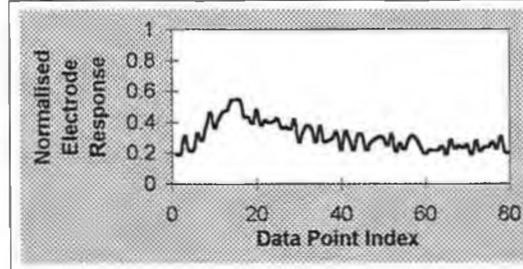


Ca ISE

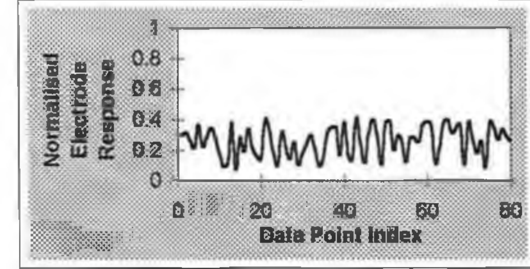
Pattern 10



Na ISE

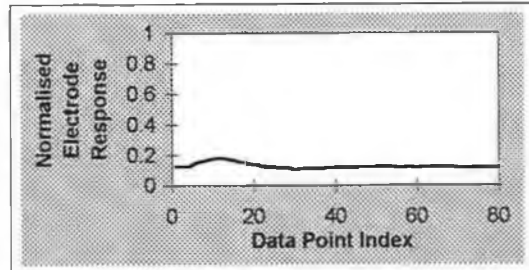


K ISE

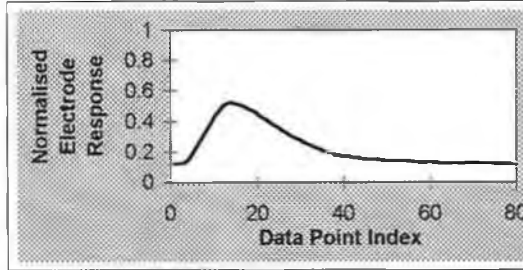


Ca ISE

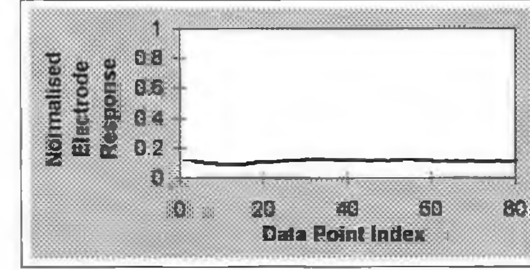
Pattern 11



Na ISE

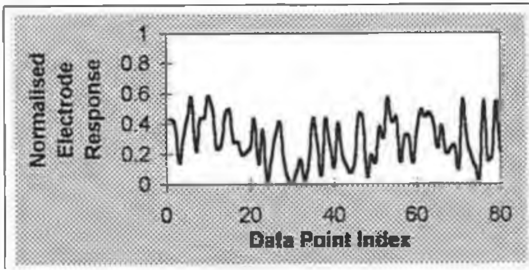


K ISE

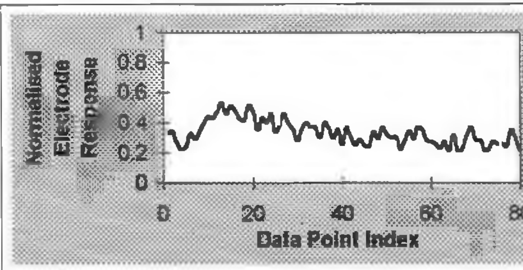


Ca ISE

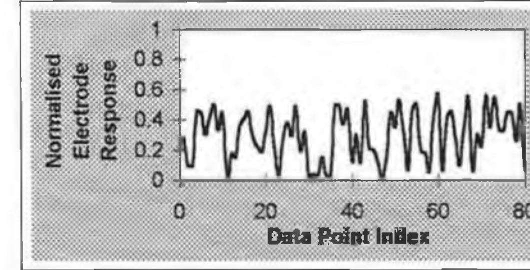
Pattern 12



Na ISE

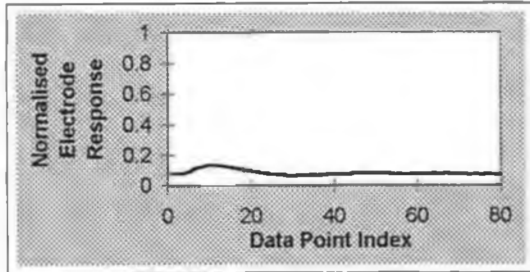


K ISE

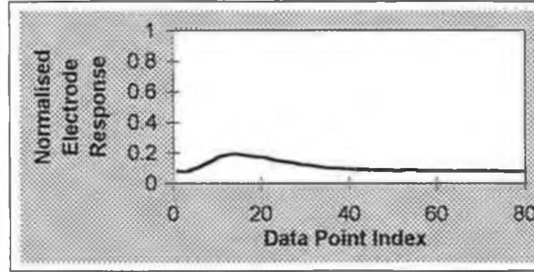


Ca ISE

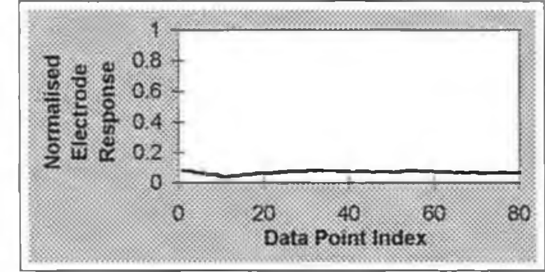
Pattern 13



Na ISE

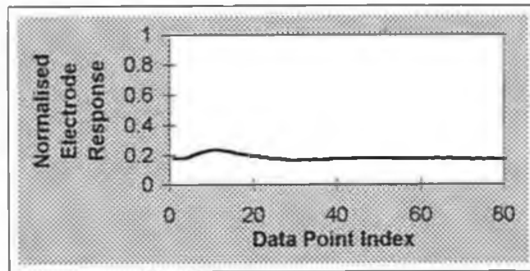


K ISE

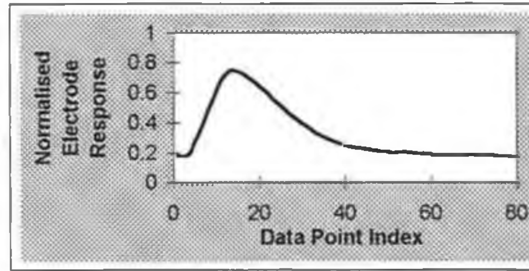


Ca ISE

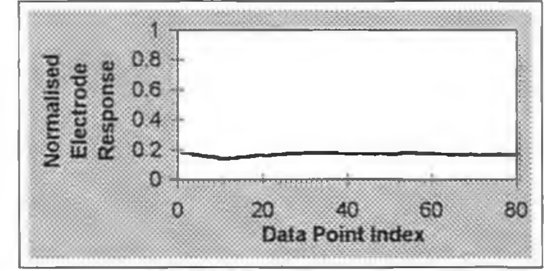
Pattern 14



Na ISE

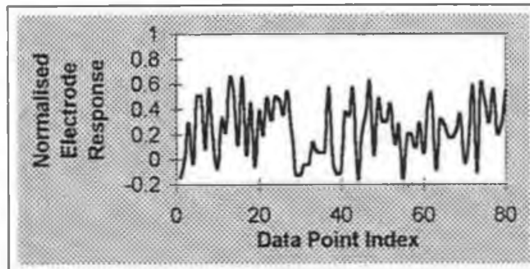


K ISE

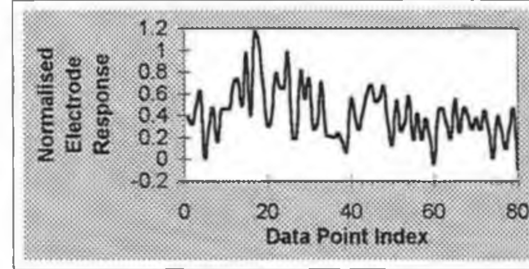


Ca ISE

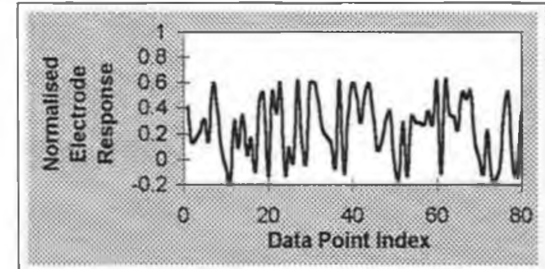
Pattern 15



Na ISE

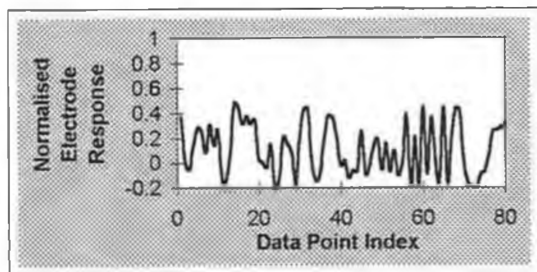


K ISE

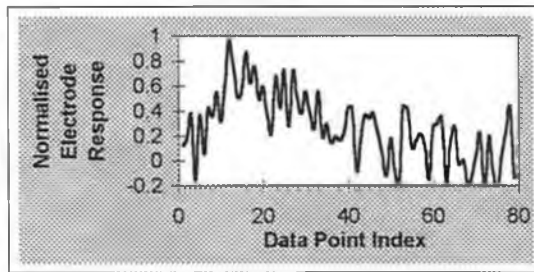


Ca ISE

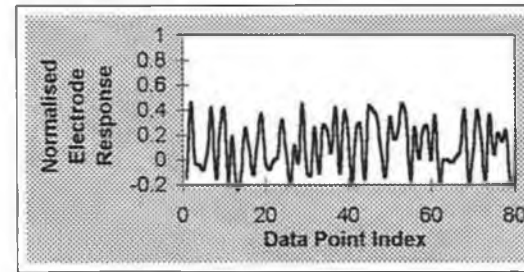
Pattern 16



Na ISE

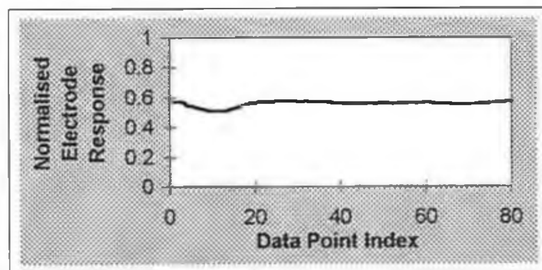


K ISE

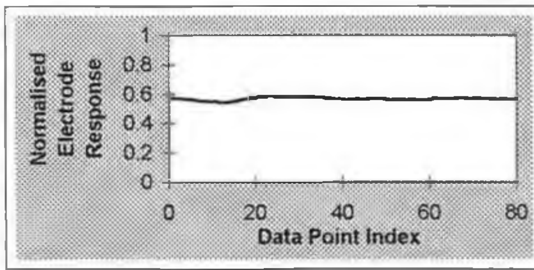


Ca ISE

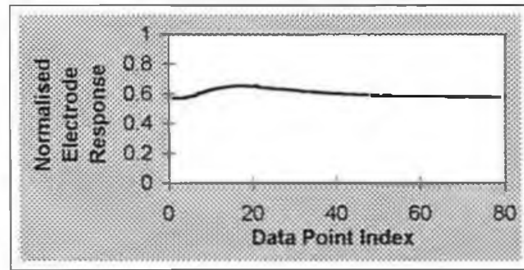
Pattern 17



Na ISE

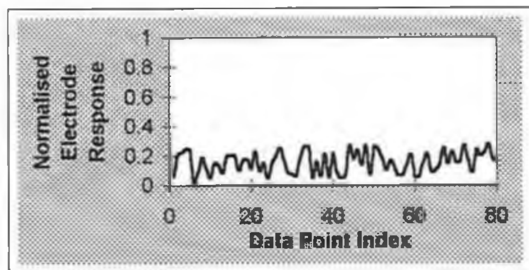


K ISE

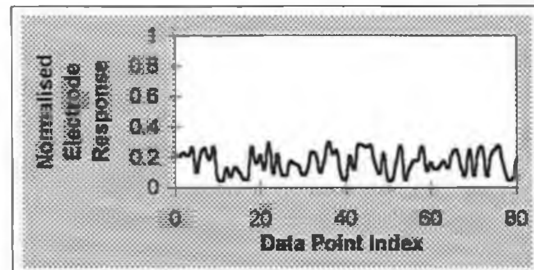


Ca ISE

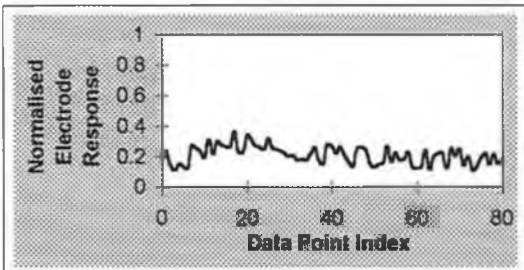
Pattern 18



Na ISE

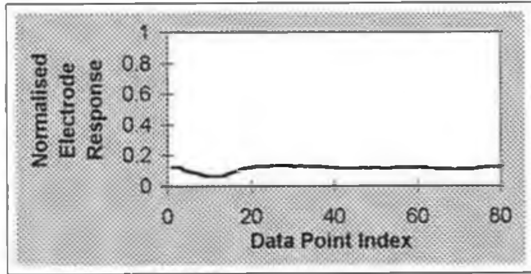


K ISE

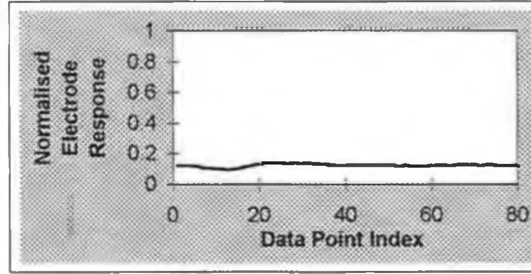


Ca ISE

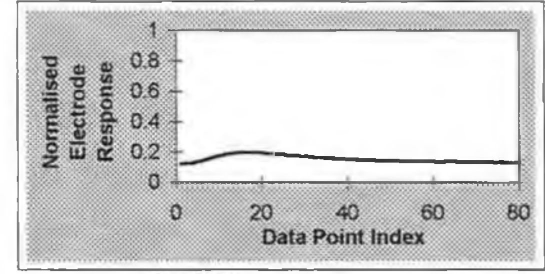
Pattern 19



Na ISE

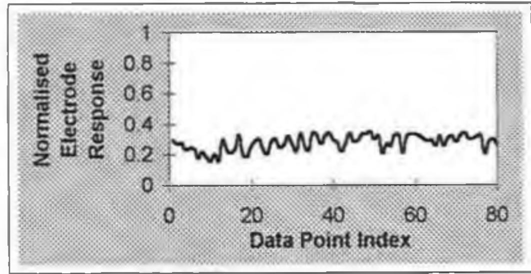


K ISE

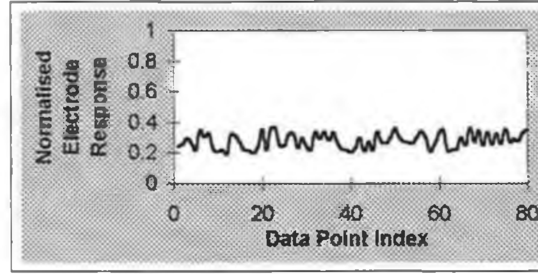


Ca ISE

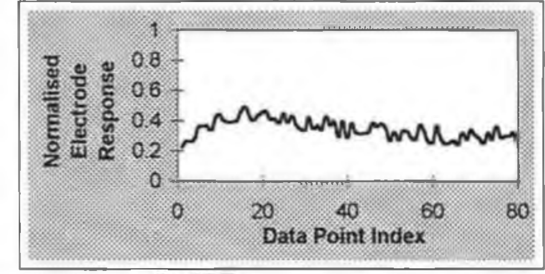
Pattern 20



Na ISE

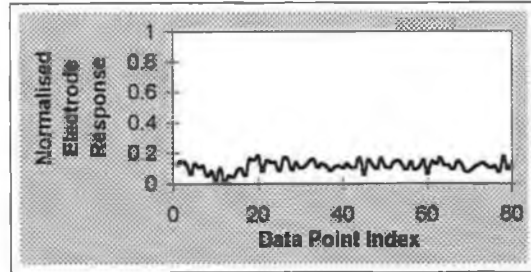


K ISE

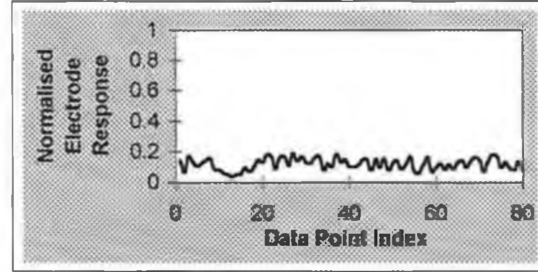


Ca ISE

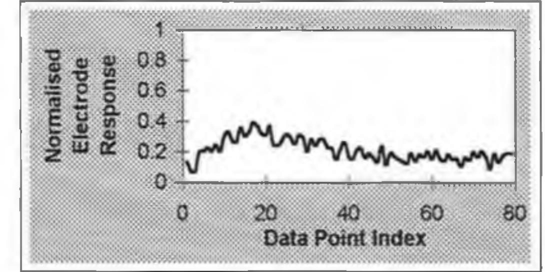
Pattern 21



Na ISE

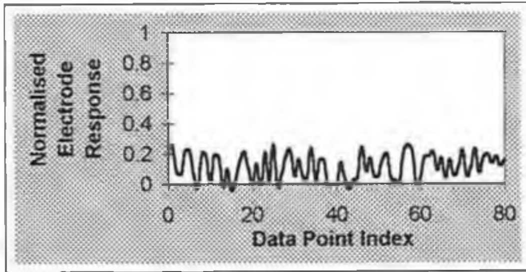


K ISE

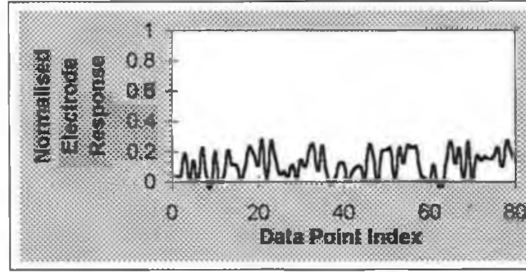


Ca ISE

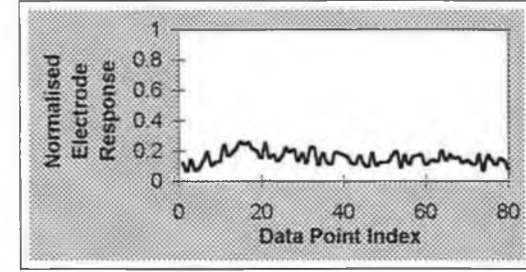
Pattern 22



Na ISE

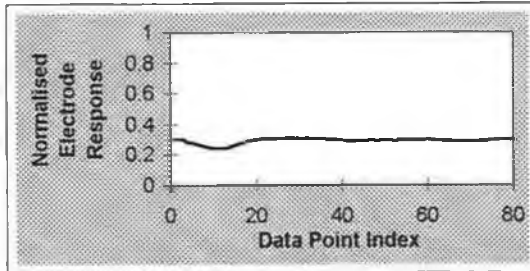


K ISE

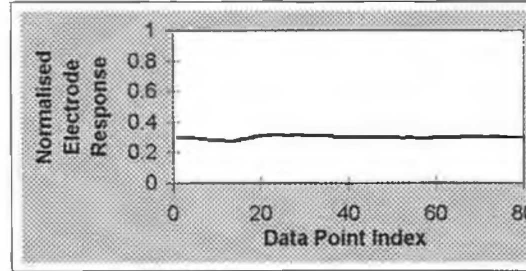


Ca ISE

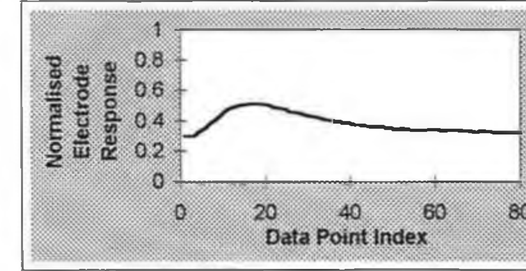
Pattern 23



Na ISE

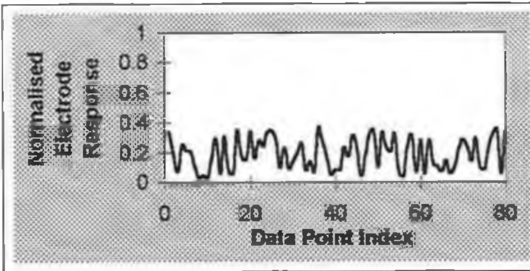


K ISE

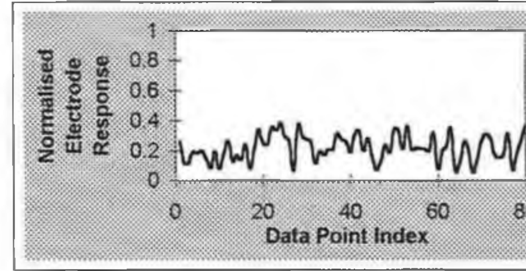


Ca ISE

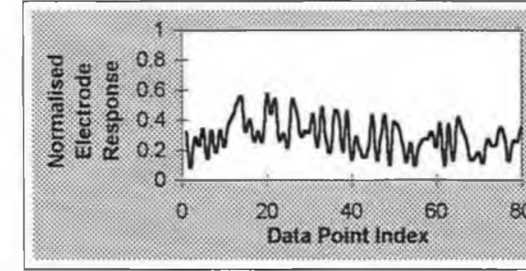
Pattern 24



Na ISE

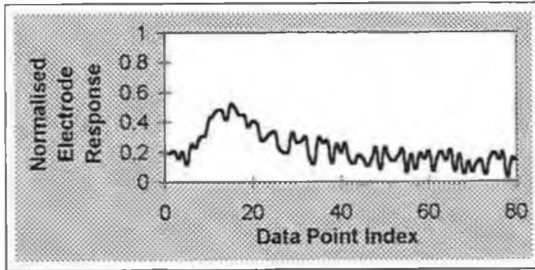


K ISE

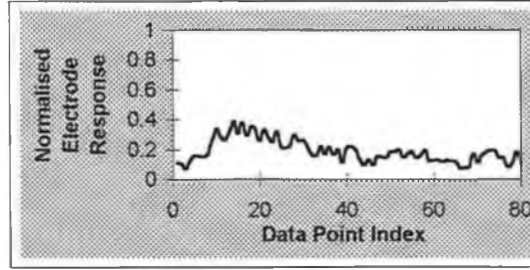


Ca ISE

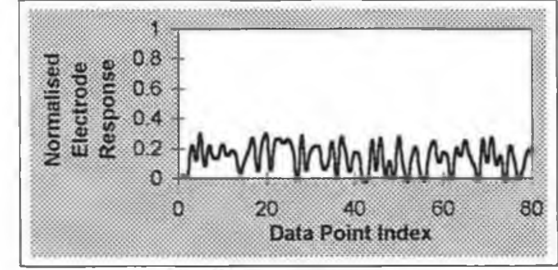
Pattern 25



Na ISE

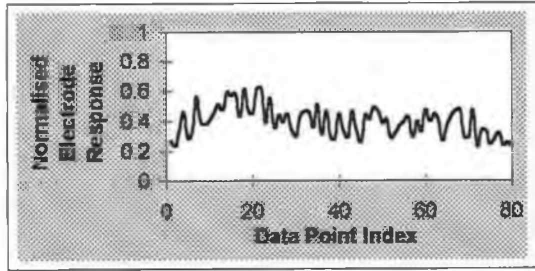


K ISE

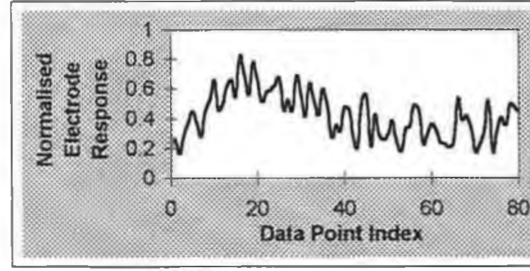


Ca ISE

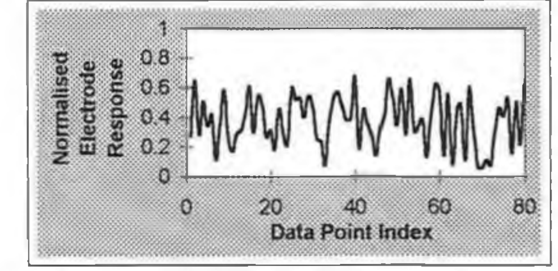
Pattern 26



Na ISE

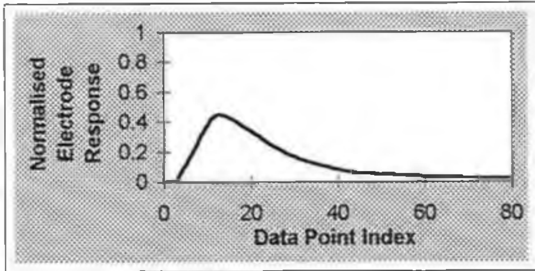


K ISE

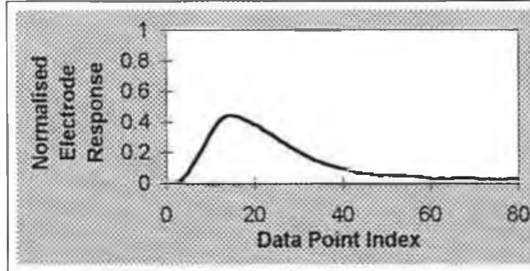


Ca ISE

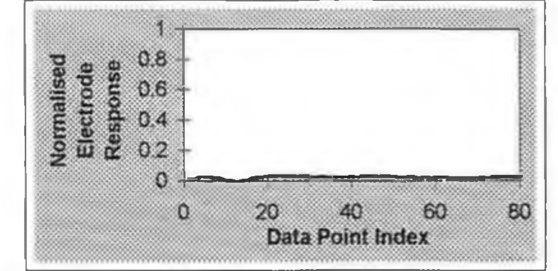
Pattern 27



Na ISE

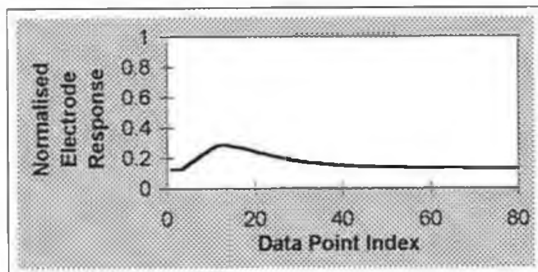


K ISE

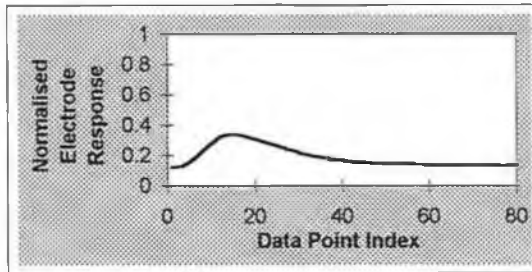


Ca ISE

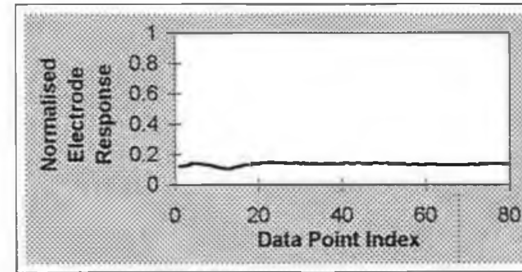
Pattern 28



Na ISE

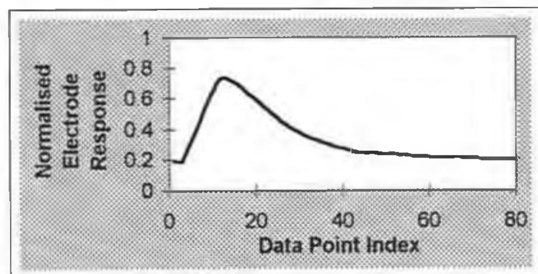


K ISE

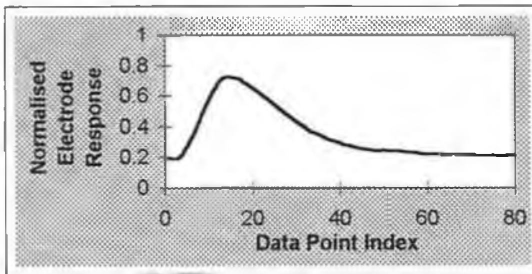


Ca ISE

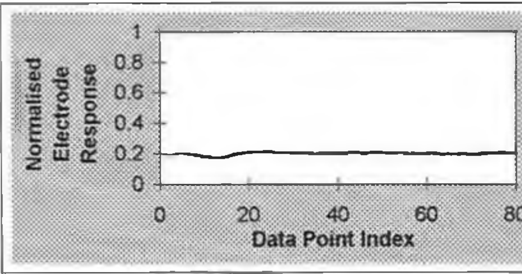
Pattern 29



Na ISE

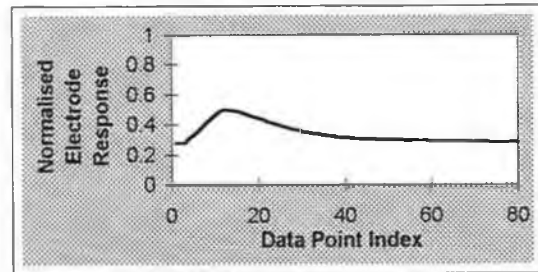


K ISE

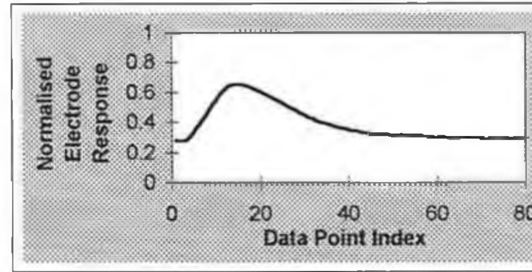


Ca ISE

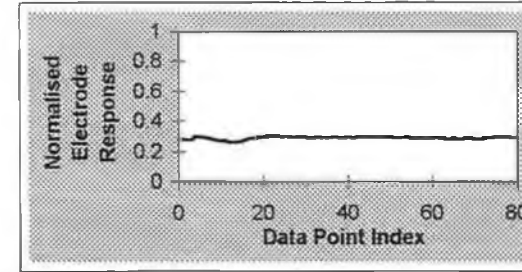
Pattern 30



Na ISE

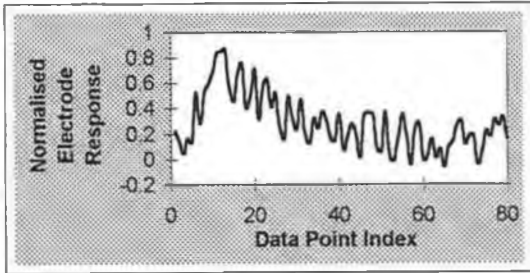


K ISE

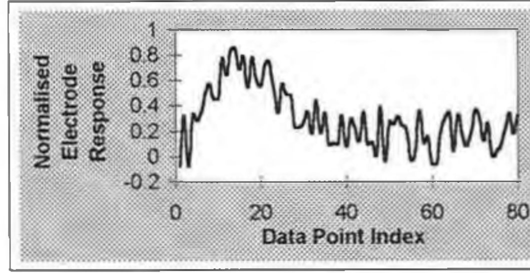


Ca ISE

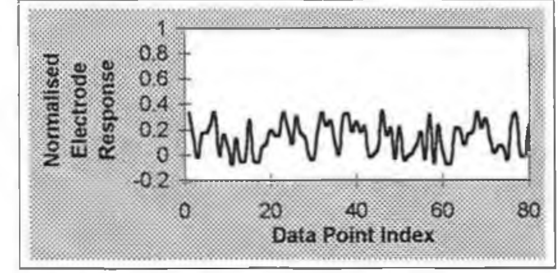
Pattern 31



Na ISE

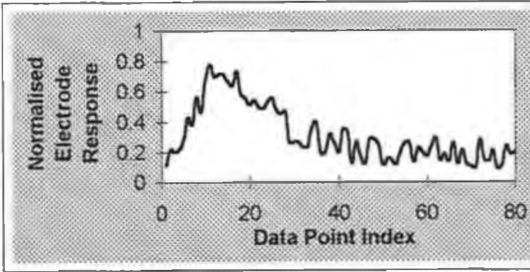


K ISE

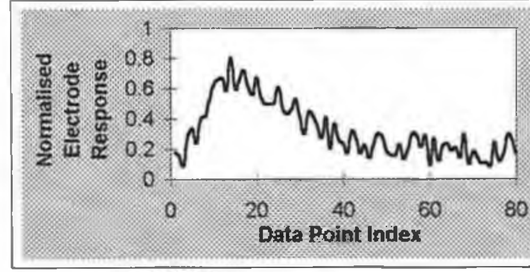


Ca ISE

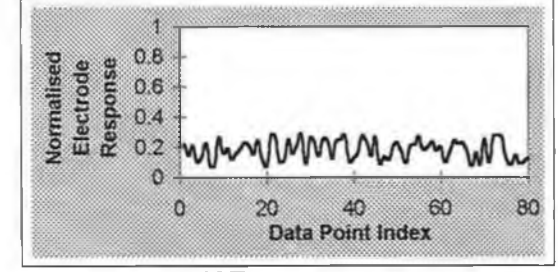
Pattern 32



Na ISE

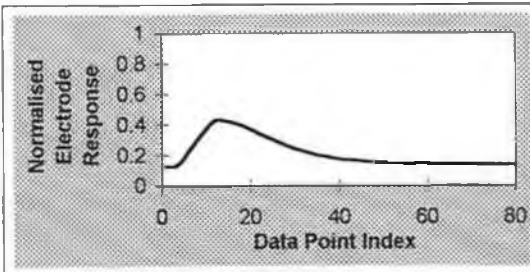


K ISE

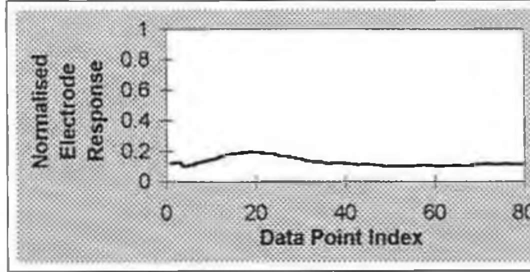


Ca ISE

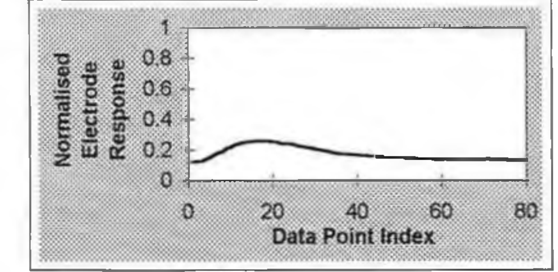
Pattern 33



Na ISE

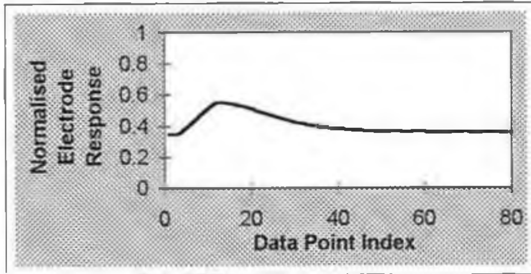


K ISE

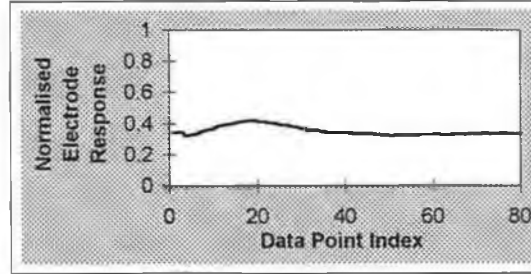


Ca ISE

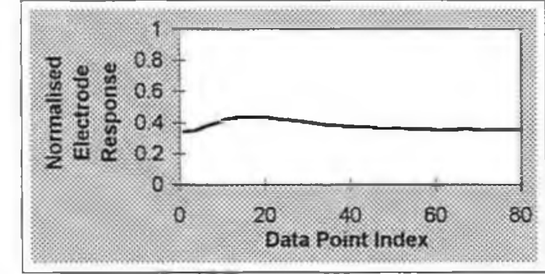
Pattern 34



Na ISE

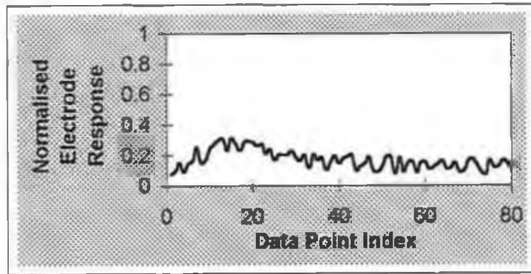


K ISE

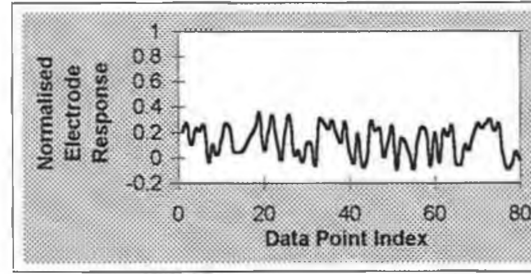


Ca ISE

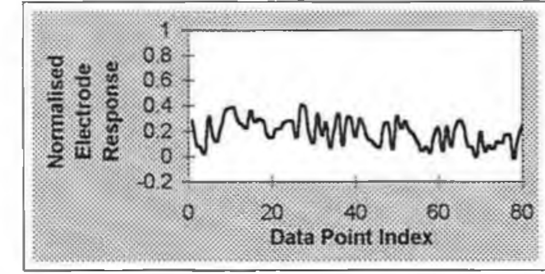
Pattern 35



Na ISE

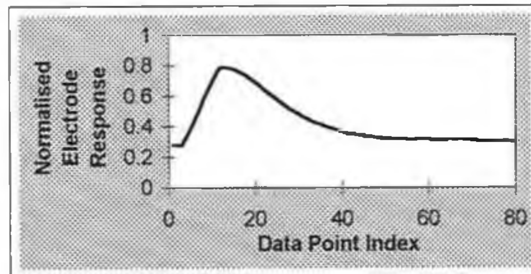


K ISE

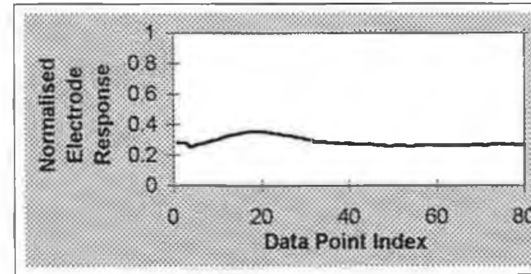


Ca ISE

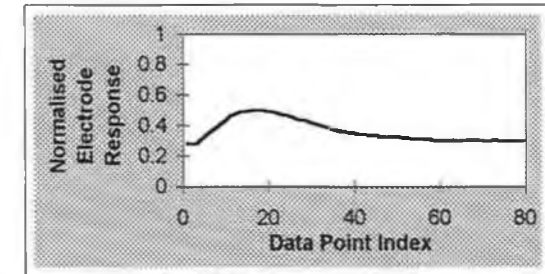
Pattern 36



Na ISE

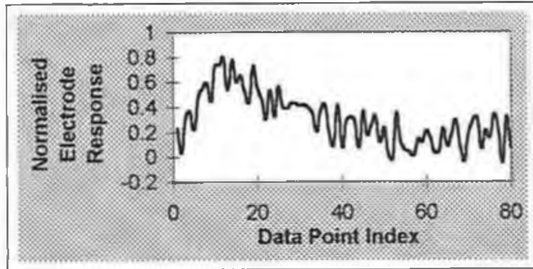


K ISE

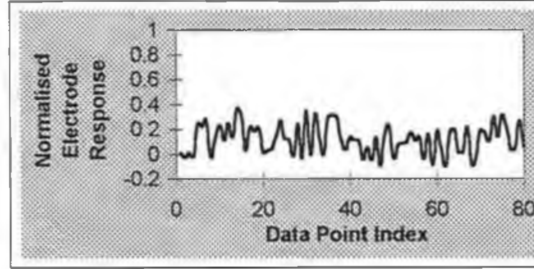


Ca ISE

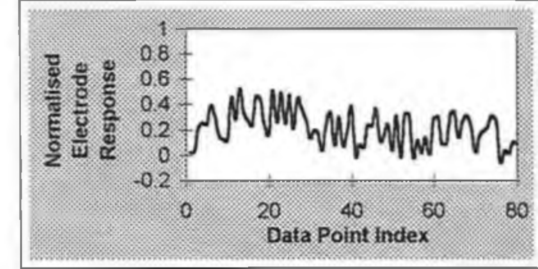
Pattern 37



Na ISE

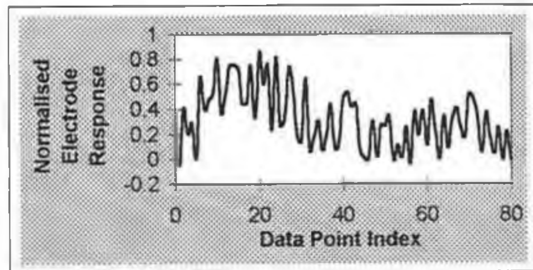


K ISE

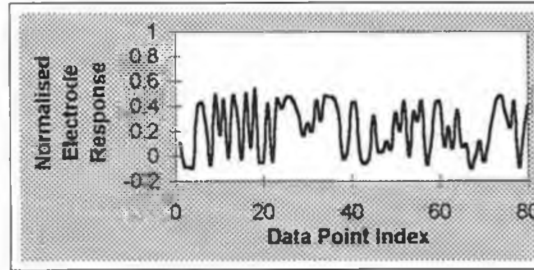


Ca ISE

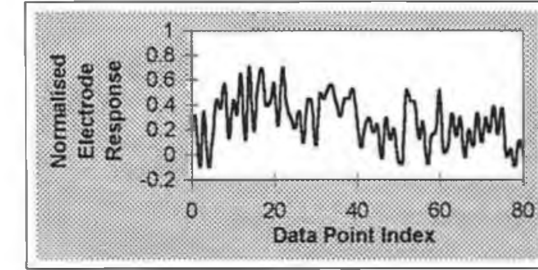
Pattern 38



Na ISE

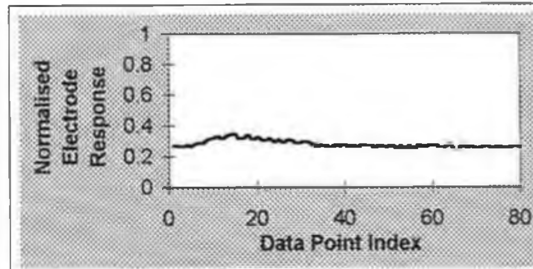


K ISE

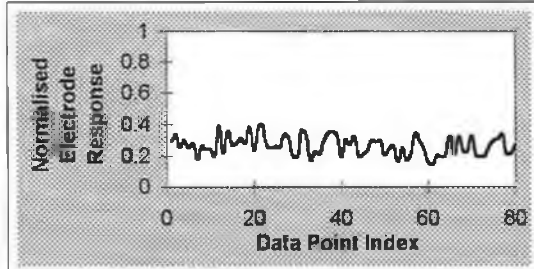


Ca ISE

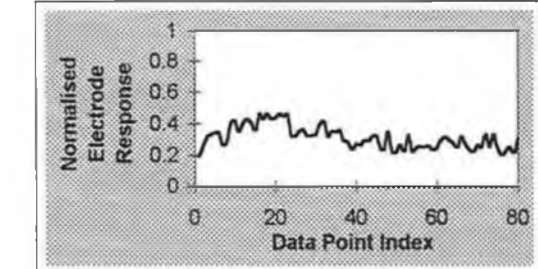
Pattern 39



Na ISE

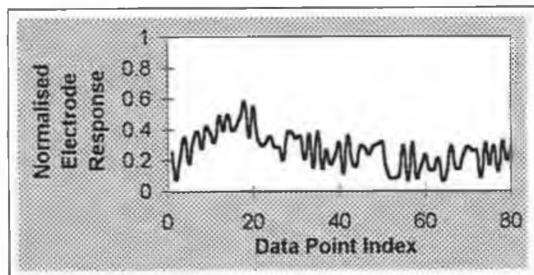


K ISE

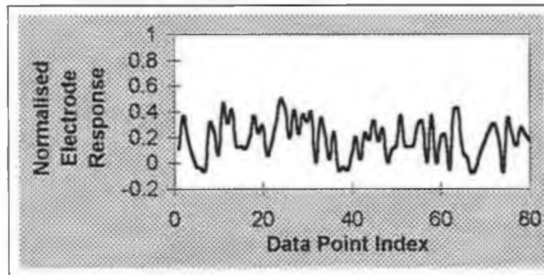


Ca ISE

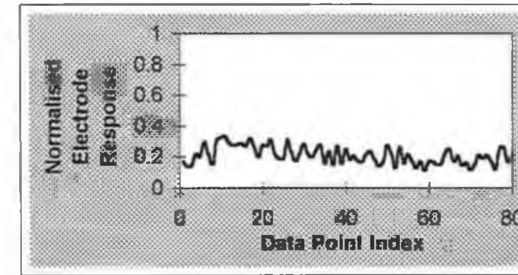
Pattern 40



Na ISE

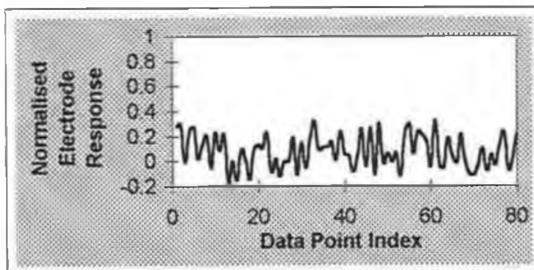


K ISE

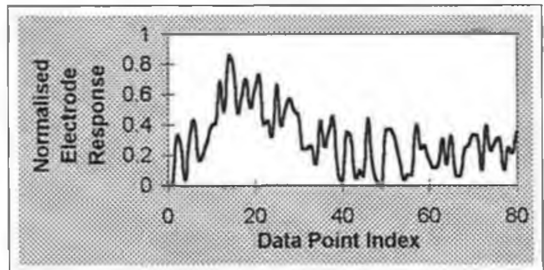


Ca ISE

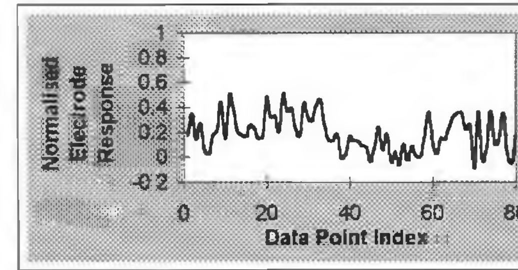
Pattern 41



Na ISE

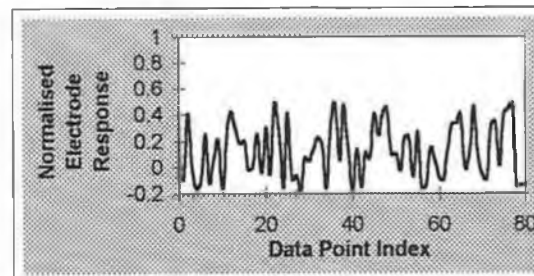


K ISE

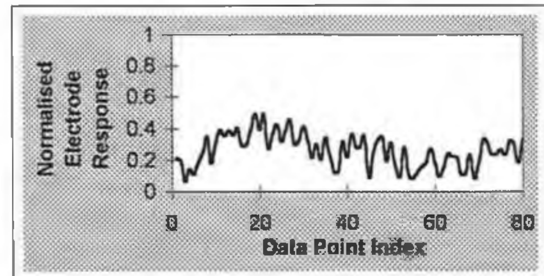


Ca ISE

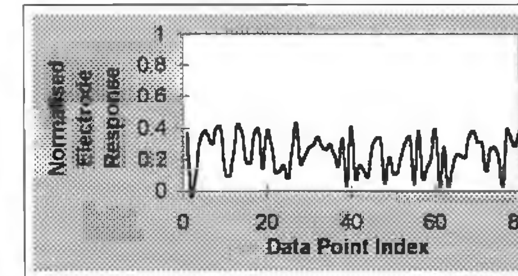
Pattern 42



Na ISE

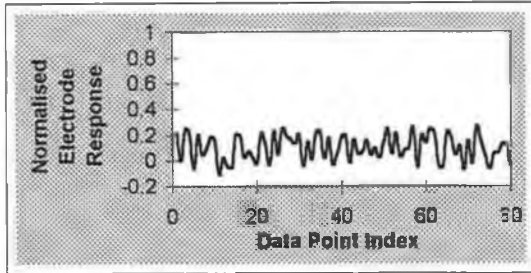


K ISE

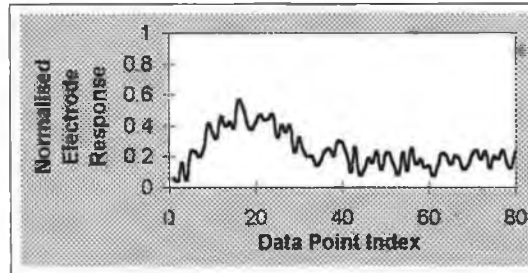


Ca ISE

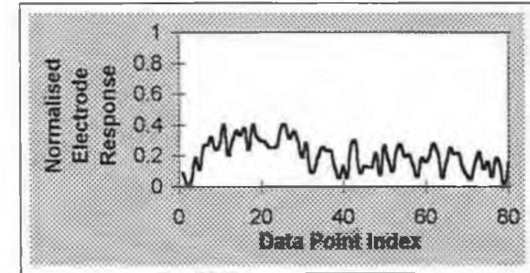
Pattern 43



Na ISE

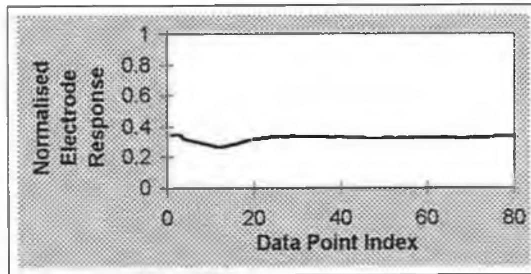


K ISE

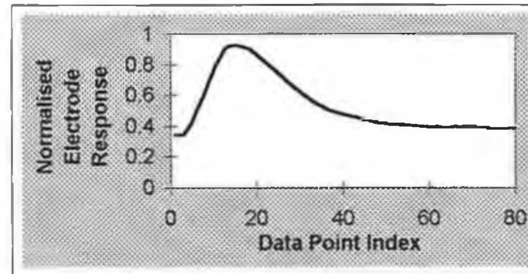


Ca ISE

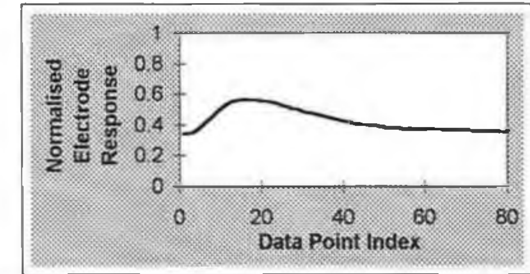
Pattern 44



Na ISE

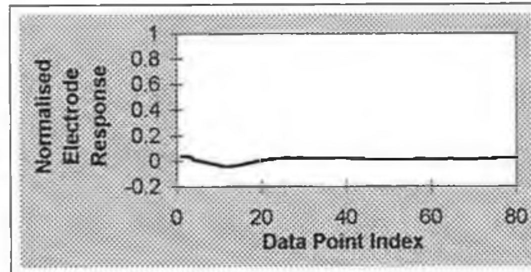


K ISE

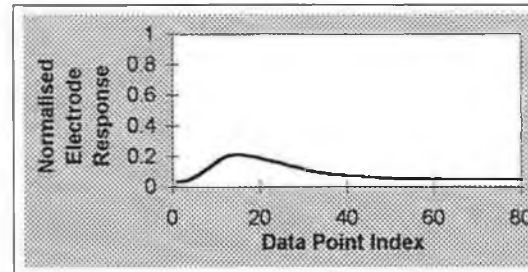


Ca ISE

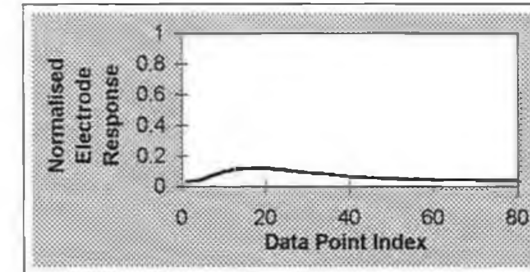
Pattern 45



Na ISE

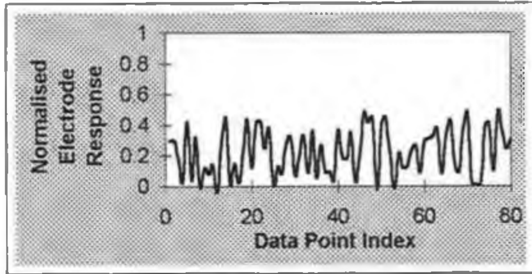


K ISE

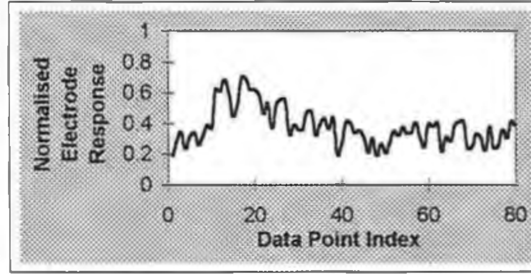


Ca ISE

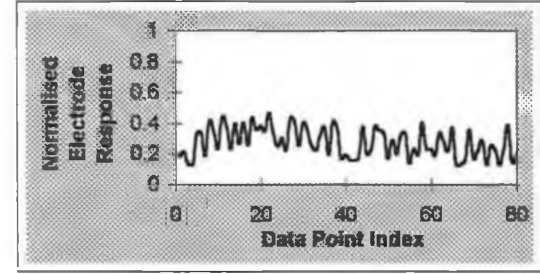
Pattern 46



Na ISE

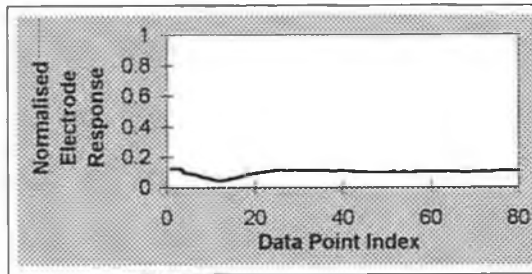


K ISE

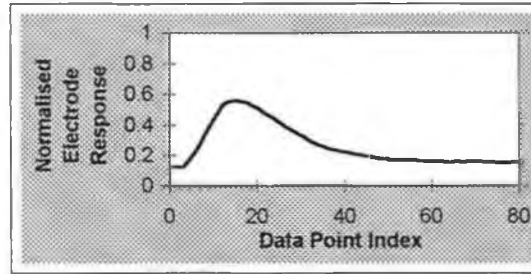


Ca ISE

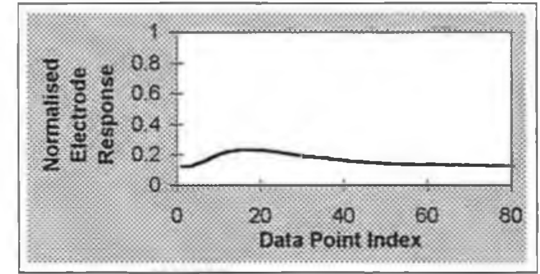
Pattern 47



Na ISE

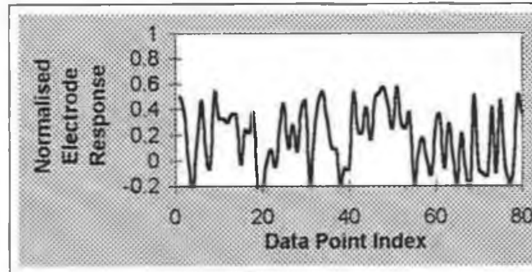


K ISE

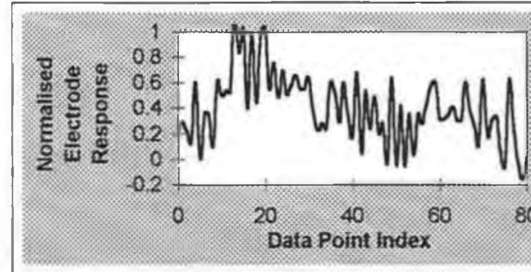


Ca ISE

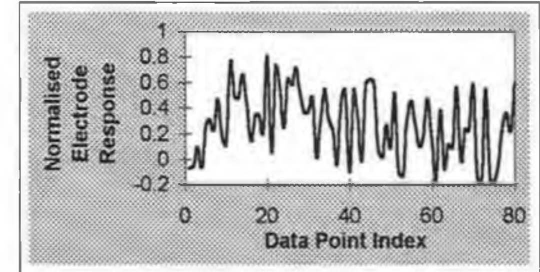
Pattern 48



Na ISE

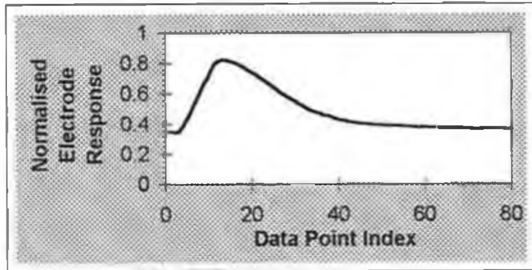


K ISE

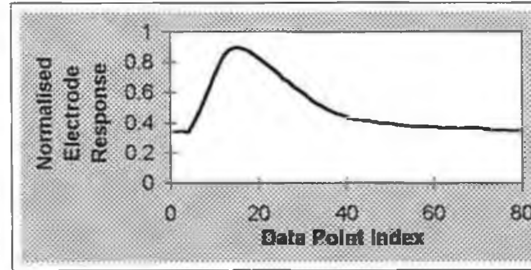


Ca ISE

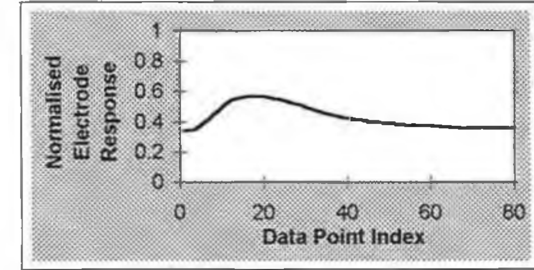
Pattern 49



Na ISE

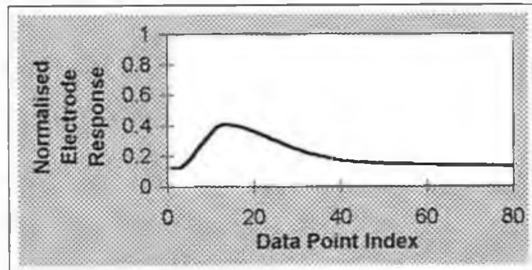


K ISE

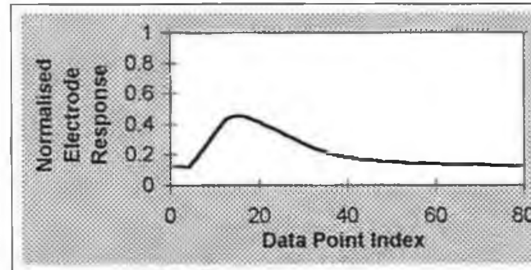


Ca ISE

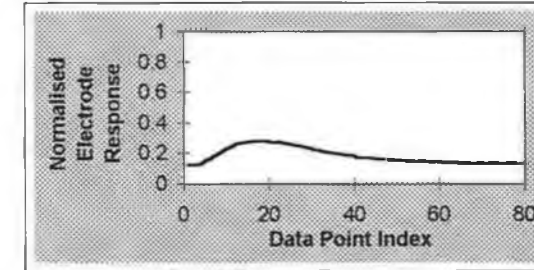
Pattern 50



Na ISE

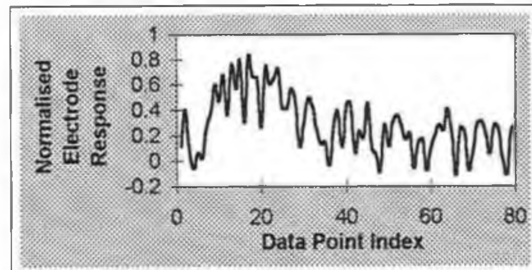


K ISE

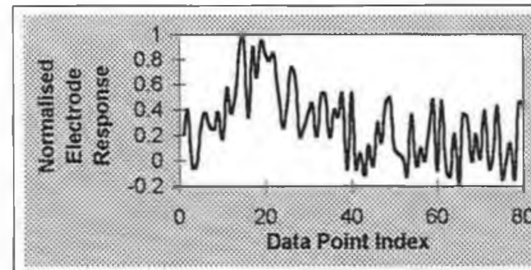


Ca ISE

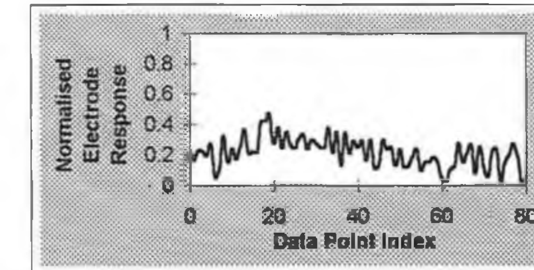
Pattern 51



Na ISE

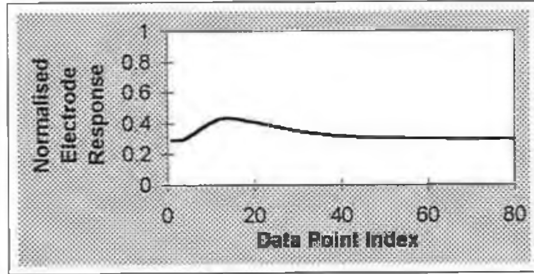


K ISE

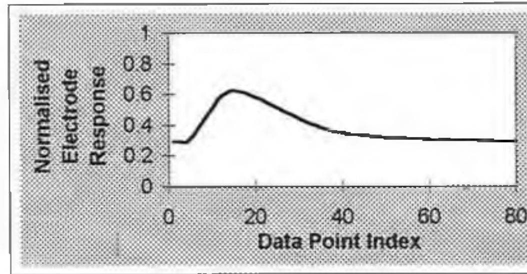


Ca ISE

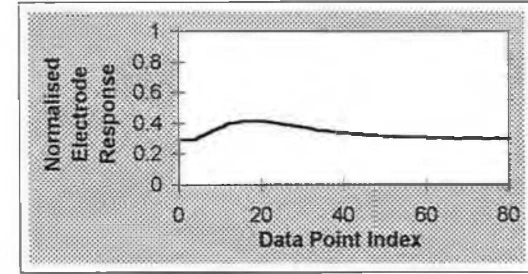
Pattern 52



Na ISE

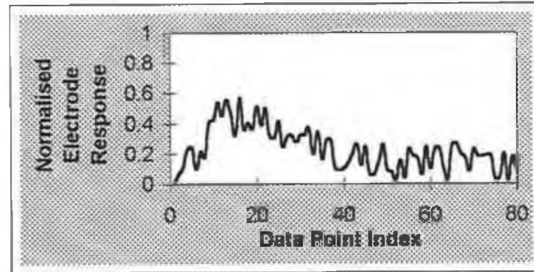


K ISE

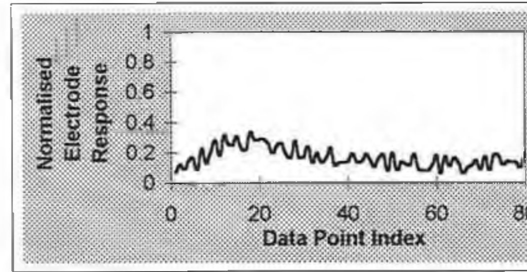


Ca ISE

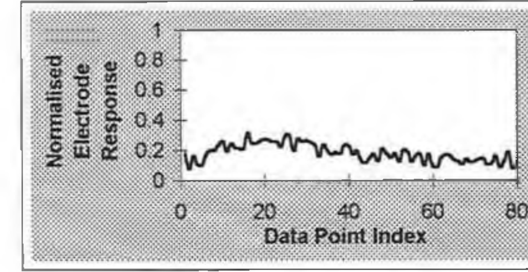
Pattern 53



Na ISE

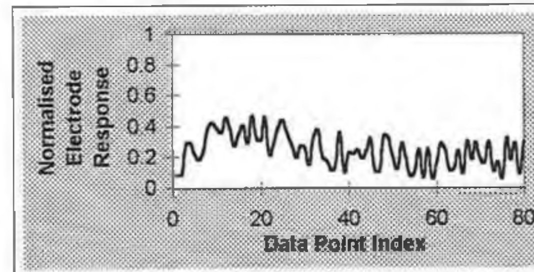


K ISE

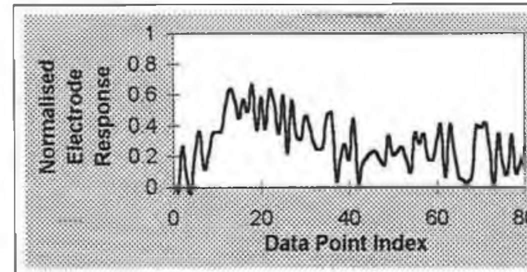


Ca ISE

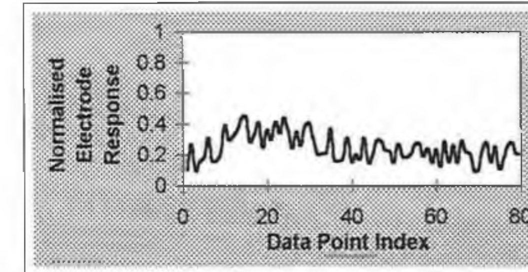
Pattern 54



Na ISE

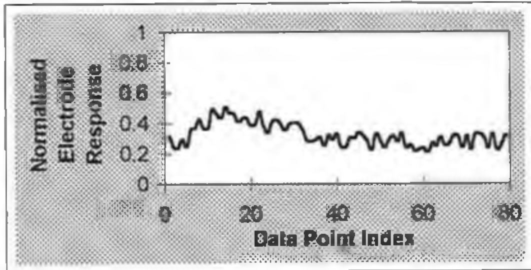


K ISE

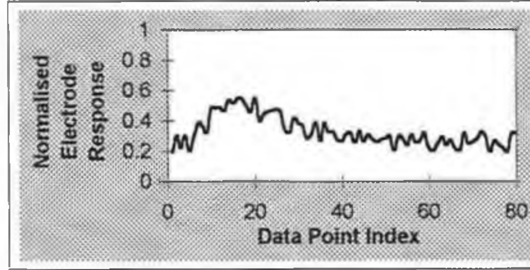


Ca ISE

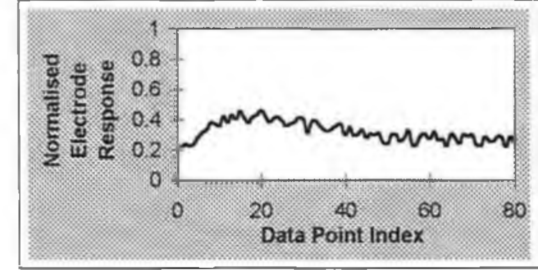
Pattern 55



Na ISE

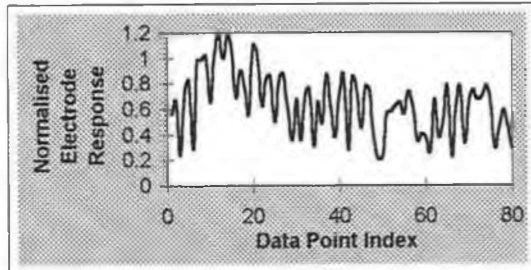


K ISE

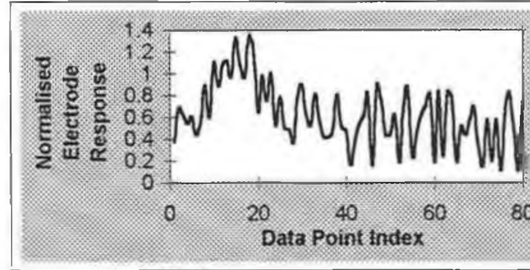


Ca ISE

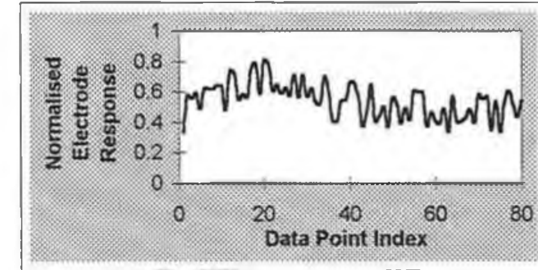
Pattern 56



Na ISE



K ISE



Ca ISE

Pattern Number	Solution Key	% Added Noise	$Na_{(rt)}$ Desired Output	$Na_{(rt)}$ Actual Output	$K_{(rt)}$ Desired Output	$K_{(rt)}$ Actual Output	$Ca_{(rt)}$ Desired Output	$Ca_{(rt)}$ Actual Output
1	Na	10	1	0.972	0	0.104	0	0.065
2	K	10	0	0.073	1	0.979	0	0.058
3	Ca	10	0	0.029	0	0.020	1	0.980
4	NaK	10	1	0.977	1	0.968	0	0.080
5	NaCa	10	1	0.983	0	0.122	1	0.957
6	KCa	10	0	0.034	1	0.973	1	0.919
7	NaKCa	10	1	0.983	1	0.936	1	0.899
8	Na	25	1	0.979	0	0.269	0	0.063
9	K	25	0	0.078	1	0.982	0	0.095
10	Ca	25	0	0.041	0	0.018	1	0.972
11	NaK	25	1	0.983	1	0.976	0	0.263
12	NaCa	25	1	0.981	0	0.152	1	0.966
13	KCa	25	0	0.041	1	0.975	1	0.912
14	NaKCa	25	1	0.979	1	0.936	1	0.926
15	Na	50	1	0.985	0	0.622	0	0.263
16	K	50	0	0.5	1	0.974	0	0.148
17	Ca	50	0	0.119	0	0.093	1	0.977
18	NaK	50	1	0.979	1	0.915	0	0.422
19	NaCa	50	1	0.961	0	0.228	1	0.960
20	KCa	50	0	0.363	1	0.718	1	0.975
21	NaKCa	50	1	0.988	1	0.893	1	0.686
22	Na	75	1	0.817	0	0.585	0	0.546
23	K	75	0	0.294	1	0.963	0	0.407
24	Ca	75	0	0.222	0	0.037	1	0.905
25	NaK	75	1	0.966	1	0.877	0	0.500
26	NaCa	75	1	0.975	0	0.508	1	0.928
27	KCa	75	0	0.148	1	0.932	1	0.844
28	NaKCa	75	1	0.844	1	0.808	1	0.826
29	Na	100	1	0.826	0	0.562	0	0.363
30	K	100	0	0.275	1	0.958	0	0.672
31	Ca	100	0	0.192	0	0.063	1	0.924
32	NaK	100	1	0.981	1	0.975	0	0.461
33	NaCa	100	1	0.984	0	0.377	1	0.835
34	KCa	100	0	0.630	1	0.867	1	0.948
35	NaKCa	100	1	0.943	1	0.954	1	0.798
36	Na	150	1	0.890	0	0.415	0	0.438
37	K	150	0	0.679	1	0.899	0	0.133
38	NaK	150	1	0.822	1	0.957	0	0.856
39	NaCa	150	1	0.922	0	0.712	1	0.718
40	KCa	150	0	0.874	1	0.887	1	0.718
41	NaKCa	150	1	0.884	1	0.969	1	0.461
42	Ca	150	0	0.187	0	0.476	1	0.798
43	Na	200	1	0.949	0	0.782	0	0.928
44	K	200	0	0.672	1	0.938	0	0.085
45	Ca	200	0	0.148	0	0.500	1	0.965
46	NaK	200	1	0.983	1	0.788	0	0.848
47	NaCa	200	1	0.919	0	0.840	1	0.743
48	KCa	200	0	0.890	1	0.593	1	0.835
49	NaKCa	200	1	0.952	1	0.905	1	0.202

Table 2.1 - Classification of patterns in appendix 2 by a network with 55 neurons in its hidden layer trained with the simple FIA patterns depicted appendix 1 to a maximum output error of 0.1, with a learning rate of 0.5 and a momentum of 0.75 (for discussion see 2.3.1.3)

			$Na_{(rt)}$			$K_{(rt)}$			$Ca_{(rt)}$		
Pattern Number	Solution Key	Peak Height Reduction	D.O.	A.O.	S/N Ratio	D.O.	A.O.	S/N Ratio	D.O.	A.O.	S/N Ratio
1	Na	10%	1	0.113	7.5	0	0.063		0	0.062	
2	K	10%	0	0.063		1	0.152	3.1	0	0.047	
3	Ca	10%	0	0.025		0	0.046		1	0.239	1.4
4	NaK	10%	1	0.101	6.5	1	0.085	6.7	0	0.104	
*	NaCa	10%									
5	KCa	10%	0	0.011		1	0.129	3.3	1	0.222	1.5
6	NaKCa	10%	1	0.104	46.3	1	0.083	57.1	1	0.119	23.6
7	Na	25%	1	0.334	18.6	0	0.056		0	0.052	
8	K	25%	0	0.060		1	0.348	7.6	0	0.037	
9	Ca	25%	0	0.022		0	0.032		1	0.445	3.2
10	NaK	25%	1	0.288	16.3	1	0.202	16.2	0	0.062	
11	NaCa	25%	1	0.400	4.3	0	0.036		1	0.314	1.9
12	KCa	25%	0	0.010		1	0.321	9.1	1	0.327	3.5
13	NaKCa	25%	1	0.288	76.6	1	0.321	89.3	1	0.327	36.7
14	Na	50%	1	0.755	37.1	0	0.049		0	0.046	
15	K	50%	0	0.056		1	0.737	15.3	0	0.032	
16	Ca	50%	0	0.022		0	0.020		1	0.803	6.3
17	NaK	50%	1	0.725	32.6	1	0.539	32.2	0	0.044	
18	NaCa	50%	1	0.813	8.5	0	0.029		1	0.615	3.8
19	KCa	50%	0	0.011		1	0.749	18.2	1	0.607	6.9
20	NaKCa	50%	1	0.793	97.2	1	0.461	113.9	1	0.492	46.2
21	Na	75%	1	0.924	55.5	0	0.058		0	0.047	
22	K	75%	0	0.058		1	0.922	22.6	0	0.033	
23	Ca	75%	0	0.026		0	0.016		1	0.945	9.3
24	NaK	75%	1	0.941	48.8	1	0.844	48.1	0	0.047	
25	NaCa	75%	1	0.948	12.8	0	0.037		1	0.856	5.2
26	KCa	75%	0	0.017		1	0.932	27.1	1	0.848	10.3
27	NaKCa	75%	1	0.966	87.7	1	0.803	102.4	1	0.822	41.5

Table 2.2 - Classification results from the network described for table 2.1 to a test set produced by reducing the heights of valid FIA peaks (the resulting test set is depicted in appendix 3 (for discussion see 2.3.1.5))

* Anomalous result, D.O.= Desired Output, A.O.= Actual Output

			$Na_{(bt)}$		$K_{(bt)}$		$Ca_{(bt)}$	
Pattern Number	Solution Key	Noise Level	Desired Output	Actual Output	Desired Output	Actual Output	Desired Output	Actual Output
1	Na	10	1	1.000	0	0.023	0	0.000
2	K	10	0	0.004	1	1.000	0	0.047
3	Ca	10	0	0.015	0	0.018	1	1.000
4	NaK	10	1	1.000	1	1.000	0	0.005
5	NaCa	10	1	1.000	0	0.018	1	1.000
6	KCa	10	0	0.000	1	1.000	1	1.000
7	NaKCa	10	1	1.000	1	1.000	1	0.999
8	Na	25	1	1.000	0	0.022	0	0.000
9	K	25	0	0.002	1	1.000	0	0.022
10	Ca	25	0	0.006	0	0.023	1	1.000
11	NaK	25	1	1.000	1	1.000	0	0.006
12	NaCa	25	1	1.000	0	0.007	1	1.000
13	KCa	25	0	0.000	1	1.000	1	1.000
14	NaKCa	25	1	1.000	1	1.000	1	0.999
15	Na	50	1	1.000	0	0.430	0	0.000
16	K	50	0	0.037	1	1.000	0	0.000
17	Ca	50	0	0.065	0	0.013	1	1.000
18	NaK	50	1	1.000	1	1.000	0	0.008
19	NaCa	50	1	1.000	0	0.062	1	1.000
20	KCa	50	0	0.001	1	1.000	1	1.000
21	NaKCa	50	1	1.000	1	1.000	1	0.989
22	Na	75	1	1.000	0	0.782	0	0.007
23	K	75	0	0.008	1	1.000	0	0.363
24	Ca	75	0	0.037	0	0.025	1	1.000
25	NaK	75	1	1.000	1	1.000	0	0.002
26	NaCa	75	1	1.000	0	0.005	1	1.000
27	KCa	75	0	0.001	1	1.000	1	1.000
28	NaKCa	75	1	1.000	1	1.000	1	0.995
29	Na	100	1	1.000	0	0.023	0	0.036
30	K	100	0	0.415	1	1.000	0	0.985
31	Ca	100	0	0.003	0	0.016	1	1.000
32	NaK	100	1	1.000	1	1.000	0	0.003
33	NaCa	100	1	1.000	0	0.000	1	1.000
34	KCa	100	0	0.152	1	1.000	1	1.000
35	NaKCa	100	1	1.000	1	1.000	1	0.993
36	Na	150	1	1.000	0	0.014	0	0.000
37	K	150	0	0.033	1	1.000	0	0.000
38	NaK	150	1	1.000	1	1.000	0	0.999
39	NaCa	150	1	1.000	0	0.002	1	0.993
40	KCa	150	0	0.835	1	1.000	1	0.996
41	NaKCa	150	1	1.000	1	1.000	1	1.000
42	Ca	150	0	0.005	0	0.160	1	1.000
43	Na	200	1	1.000	0	0.998	0	1.000
44	K	200	0	0.341	1	1.000	0	0.002
45	Ca	200	0	0.010	0	0.002	1	1.000
46	NaK	200	1	1.000	1	0.994	0	0.919
47	NaCa	200	1	1.000	0	0.863	1	0.056
48	KCa	200	0	0.997	1	0.994	1	0.870
49	NaKCa	200	1	1.000	1	1.000	1	0.046

Table 2.3 - Classification of patterns in appendix 2 by a network with 55 neurons in its hidden layer trained with the distorted FIA patterns depicted appendix 5 to a maximum output error of 0.1, with a learning rate of 0.5 and a momentum of 0.75 (for discussion see 2.3.3.2 test 1)

			$Na_{(bt)}$			$K_{(bt)}$			$Ca_{(bt)}$		
Pattern Number	Solution Key	Peak Height Reduction	D.O.	A.O.	S/N Ratio	D.O.	A.O.	S/N Ratio	D.O.	A.O.	S/N Ratio
1	Na	10%	1	0.912	7.5	0	0.018		0	0.003	
2	K	10%	0	0.355		1	0.915	3.1	0	0.004	
3	Ca	10%	0	0.052		0	0.052		1	0.554	1.4
4	NaK	10%	1	0.699	6.5	1	0.484	6.7	0	0.021	
*	NaCa	10%									
5	KCa	10%	0	0.002		1	0.977	3.3	1	0.607	1.5
6	NaKCa	10%	1	0.692	46.3	1	0.492	57.1	1	0.093	23.6
7	Na	25%	1	0.998	18.6	0	0.006		0	0.001	
8	K	25%	0	0.098		1	1.000	7.6	0	0.007	
9	Ca	25%	0	0.038		0	0.027		1	0.928	3.2
10	NaK	25%	1	0.991	16.3	1	0.987	16.2	0	0.013	
11	NaCa	25%	1	0.999	4.3	0	0.011		1	0.637	1.9
12	KCa	25%	0	0.001		1	1.000	9.1	1	0.938	3.5
13	NaKCa	25%	1	0.991	76.6	1	0.984	89.3	1	0.607	36.7
14	Na	50%	1	1.000	37.1	0	0.008		0	0.000	
15	K	50%	0	0.018		1	1.000	15.3	0	0.015	
16	Ca	50%	0	0.023		0	0.015		1	0.997	6.3
17	NaK	50%	1	1.000	32.6	1	1.000	32.2	0	0.021	
18	NaCa	50%	1	1.000	8.5	0	0.03		1	0.986	3.8
19	KCa	50%	0	0.001		1	1.000	18.2	1	0.998	6.9
20	NaKCa	50%	1	1.000	97.2	1	1.000	113.9	1	0.990	46.2
21	Na	75%	1	1.000	55.5	0	0.017		0	0.000	
22	K	75%	0	0.007		1	1.000	22.6	0	0.028	
23	Ca	75%	0	0.017		0	0.015		1	1.000	9.3
24	NaK	75%	1	1.000	48.8	1	1.000	48.1	0	0.010	
25	NaCa	75%	1	1.000	12.8	0	0.012		1	0.999	5.2
26	KCa	75%	0	0.001		1	1.000	27.1	1	1.000	10.3
27	NaKCa	75%	1	1.000	87.7	1	1.000	102.4	1	0.998	41.5

Table 2.4 - Classification results from the network described for table 2.3 to a test set produced by reducing the heights of valid FIA peaks (the resulting test set is depicted in appendix 3 (for discussion see 2.3.2.2 test 2))

* Anomalous result, D.O.= Desired Output, A.O.= Actual Output

Solution	NH_4^+ activity	Na^+ activity	K^+ activity	Ca^{2+} activity	Solution	NH_4^+ activity	Na^+ activity	K^+ activity	Ca^{2+} activity
1	0.007887	0.007098	0.002366	0.003183	17	0.006262	0.006262	0.007827	0.003093
2	0.008463	0.002539	0.000423	0.000468	18	0.000741	0.000494	0.008228	0.002336
3	0.008416	0.000842	0.004208	0.000356	19	0.003378	0.000676	0.008446	0.000361
4	0.008368	0.000418	0.000502	0.001495	20	0.00068	0.002548	0.008495	0.000211
5	8.57E-05	0.005998	0.003428	0.000218	21	0.001632	0.000734	8.16E-05	0.004072
6	8.58E-05	0.00429	0.000686	0.001097	22	0.000256	0.003415	8.54E-05	0.001614
7	8.34E-05	0.000167	0.002502	0.002952	23	0.000868	0.004338	8.68E-05	0.000515
8	8.96E-05	0.000269	0.000179	0.00013	24	0.000179	0.000358	8.95E-05	0.000193
9	0.007058	0.007842	0.007058	0.002726	25	0.006312	0.005523	0.000631	0.003984
10	0.000756	0.008401	0.005041	0.000303	26	0.003961	0.000555	0.006338	0.004047
11	0.000332	0.0083	0.000581	0.001932	27	0.000244	0.000813	0.000325	0.004457
12	0.002538	0.008461	0.000761	0.000416	28	0.000569	8.12E-05	0.000812	0.004453
13	0.000421	8.43E-05	0.001686	0.00256	29	0.005129	0.005129	0.000598	5.41E-05
14	0.00508	8.47E-05	0.005926	0.000521	30	0.002619	0.000699	0.002619	5.87E-05
15	0.005774	8.25E-05	0.00033	0.00283	31	0.000704	0.001761	0.001761	6.05E-05
16	0.000178	8.92E-05	0.000268	0.000319	32	0.000537	0.000448	8.95E-05	6.46E-05

Table 3.1 - Activities of ammonium, sodium, potassium and calcium in the calibration solutions in chapter 3(corresponding to the concentrations seen in table 3.1 of chapter 3).

Appendix 8: Rank Scaling For GA In Chapter 3 [See 3.5.1.1]

$\overline{Err(n-1)}$ =Average Error

$\overline{Err(n)}$ =Prescaled Average Error

Err(min,n-1)= Lowest Error In Current Population

Err(min,n)=Rescaled Lowest Error In Current Population

let α= prescaling constant or maximum number of progeny allowed to individual with lowest error

$$\overline{Err(n)} = \overline{Err(n-1)} \qquad \text{(Condition 2)}$$

$$\frac{\overline{Err(n)}}{Err(\min,n)} = \alpha \Rightarrow Err(\min,n) = \frac{1}{\alpha}\overline{Err(n)} \qquad \text{(Condition 1)}$$

$$\overline{Err(n)} = m.\overline{Err(n-1)} + c \qquad Err(\min,n) = m.Err(\min,n-1) + c$$

$$m = \frac{\overline{Err(n-1)} - \frac{1}{\alpha}\overline{Err(n)}}{\overline{Err(n-1)} - Err(\min,n)}$$

$$c = Err(\min,n) - m.Err(\min,n-1) = \overline{Err(n-1)}\left(\frac{\frac{1}{\alpha}\overline{Err(n-1)} - Err(\min,n-1)}{\overline{Err(n-1)} - Err(\min,n-1)}\right)$$

$$Err(n) = \frac{\overline{Err(n-1)} - \frac{1}{\alpha}\overline{Err(n)}}{\overline{Err(n-1)} - Err(\min,n)}.Err(n-1) + \overline{Err(n-1)}\left(\frac{\frac{1}{\alpha}\overline{Err(n-1)} - Err(\min,n-1)}{\overline{Err(n-1)} - Err(\min,n-1)}\right)$$

Randhead.c

```
/***************************************************************/
/*                                                             */
/*          RANDOM FUNCTION INCLUDE FILE                       */
/*          RANDHEAD.C                                         */
/* This file contains the random number functions which will be used with */
/* the genetic algorithm to replace the system random number generator . */
/* These routines are a translation of the routines described in Goldberg's */
/* book(cf. Donald E. Knuth) - semi-numerical algorithms.      */
/*                                                             */
/* The routines are composed as follows :                      */
/*                                                             */
/*     warmup : initialises the random number generator        */
/*     VOID WARMUP(FLOAT RANDOMSEED)                           */
/*          CALLS: ADVANCE() X 3                               */
/*                                                             */
/*     advance : generates a new batch of pseudo-random numbers */
/*     VOID ADVANCE(VOID)                                      */
/*          CALLS: NONE                                        */
/*                                                             */
/*     randnew: returns a SINGLE pseudo-random real value      */
/*     FLOAT RANDNEW(VOID)                                     */
/*          CALLS: ADVANCE()                                   */
/*                                                             */
/*     flip : returns result of simulated coin toss, Boolean (true/false) */
/*     INT FLIP(FLOAT PROBABILITY)                             */
/*          CALLS: RANDNEW()                                   */
/*                                                             */
/*     randnew: fetch a single random number between 0 and 1   */
/*     (subtractive method)                                    */
/*     FLOAT RANDNEW(VOID)                                     */
/*          CALLS: ADVANCE()                                   */
/*                                                             */
/*     pickrand: picking a random INT between the low and high ranges */
/*     INT PICKRAND(INT LOW,INT HIGH)                          */
/*          CALLS: RANDNEW()                                   */
/*                                                             */
/*     setrand: get seed number for RANDNEW to start random number */
/*     generator                                               */
/*     VOID SETRAND(VOID)                                      */
/*          CALLS: WARMUP(SEED)                                */
/*                                                             */
/*                                                             */
/*     Margaret Hartnett                                       */
/*     1993                                                    */
/***************************************************************/

#include <time.h>
#include <dos.h>
```

```
/* WARNING GLOBAL VARIABLES !!! DON'T USE THESE IN CALLING CODE
*/

float oldrand[56];   /*** array of 55 random numbers          ***/
int jrand;           /*** index of current random number  ***/


/**** FUNCTION CALLS ****/
void advance(void);
void warmup(float randomseed);
int flip(float probability);
float randnew(void);
int pickrand(int low, int high);
void setrand(void);

void advance(void)
 {
 int ji;
 float newrandom;

 for(ji=1;ji<=24;ji++)
  {
        newrandom = oldrand[ji] - oldrand[ji+31];
        if(newrandom < 0.0)
              newrandom= newrandom +1.0;
        oldrand[ji] = newrandom;
  }

 for(ji=25;ji<=55;ji++)
  {
        newrandom = oldrand[ji] - oldrand[ji-24];
        if(newrandom < 0.0)
              newrandom = newrandom + 1.0;
        oldrand[ji] = newrandom;
  }

 }  /**** end of advance function ****/


 void warmup(float randomseed)
  {
   int ji, ii;
   float newrandom, prevrand;

   oldrand[55] = randomseed;
   newrandom = 1e-9;
   prevrand= randomseed;

   for(ji=1;ji<=54;ji++)
    {
```

```
            ii = (21*ji)%55;
            oldrand[ii] = newrandom;
            newrandom = prevrand-newrandom;
            if(newrandom < 0.0)
               newrandom = newrandom + 1.0;
            prevrand = oldrand[ii];
        }
      advance();
      advance();
      advance();
      jrand =0;
    }/** end of warmup function **/


int flip(float probability)
   {
    int toss;
    if(probability == 1.0)
         toss =1;
    else
         {
          if(randnew()<=probability)
            toss =1;
          else toss =0;
         }
    return(toss);
   }/** end of flip function **/


  float randnew(void)
   {
      float randval;
        /*** fetching a single random number between 0.0 and 1.0 ***/
        /*** subtractive method                                  ***/

        jrand = jrand +1;
        if(jrand > 55)
          {
           jrand = 1;
           advance();
          }

        randval = oldrand[jrand];
        return(randval);
    } /** end of randnew function **/


int pickrand(int low, int high)
    {
     /*** picking a random INTEGER between the low and high  ***/
```

```
  /*** ranges specified by the function call                    ***/

  int i;
  double itemp;

  if(low >=high)
       i = low;
  else
      {
       itemp = randnew()*(high-low + 1.0)+ low;
       i =(int)floor(itemp);
       if(i > high)
          i = high;
      }
  return(i);
 } /** end of function pickrand **?


void setrand(void)
 {
 /*** get seed number for RANDNEW to start random number generation ***/
 float seed;
 double tim2,timsec;
 double temptim;
 time_t t;
 long days=86400;
 long remder;

 t=time(NULL);
 remder=t%days;
 tim2 = log10((double)remder);
 tim2 =ceil(tim2);
 seed(float)((double)remder/pow(10.0,tim2));
  printf("seed %f\n",seed);
 warmup(seed);
 } /** end of setrand function **/
```

```
/************************************************************************/
/*      GENETIC ALGORITHM FOR MULTIVARIATE CALIBRATION                   */
/*                                                                       */
/*      The preliminary code for the simple GA was a translation of PASCAL */
/*      software written by Boris Fennema.                               */
/*      This software performs the 3 operations of reproduction, mutation and */
/*      crossover.                                                       */
/*                                                                       */
/*              REPRODUCTION                                             */
/*      Reproduction probability is based on the chi squared objective function */
/*      ( non-linear scaling may be implemented later).                  */
/*                                                                       */
/*      The random function header file will be included to allow for full range of */
/*      random/pseudorandom functions.                                   */
/*                                                                       */
/*      A quick sort is used to find the best fitting individual in the population, */
/*      the sort is performed in terms of the indices of the chromosomes rather */
/*      than on the chromosomes themselves.                              */
/*                                                                       */
/*                                                                       */
/*      Breeding will be performed using stochastic remainder selection based on */
/*      rank prescaling rather than fitness prescaling                   */
/*                                                                       */
/*                                                                       */
/*              MUTATION                                                 */
/*     The XOR function is employed for toggling due to the mutation operator. */
/*                                                                       */
/*              CROSSOVER                                                */
/*      Indices of both the genes and of the allele positions are used determining */
/*      crossover points.  A single crossover point approach is used.    */
/*                                                                       */
/************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <alloc.h>
#include <dos.h>
#include <string.h>
#include "randhead.c"

#define MAXGENES 500
#define MAXPARM 30
#define MAXCYCLES 2000
#define MAXWORD 65535
#define WORDSIZE 16
#define MAXFIT 1E100
#define THRESHOLD 1E-3
#define CROSSRATE 0.65
```

```
#define NUMPARM 5 /**** CHANGE THIS LATER BACK TO 20 ****/
#define RANDSIZE  1000

        void prescale(void);
          void filhand(void);
          void sort(void);
          void qsortc(int l,int r);
          void fillgene(void);
          void fillpool(void);
          void breedgene(void);
          void mutategene(void);
          void crossgenes(void);
          long int doubletoword(double value,int index);
          void evalgenes(void);
          void stat(void);
          double getchromosome(int geneindex,int chromindex);
          void select(void);
          void rankscale(void);
          void preselect(void);

int chromosome[MAXPARM];  /*THE LENGTH OF THE CHROMOSOME IS */
                          /*DETERMINED BY THE NUMBER OF      */
                          /* PARAMETERS TO BE OPTIMISED FOR*/
long int far *genepool[402];
int numdata,numgenes,numcycles,crosspool;
float anh[200],ana[200],ak[200],aca[200];
float enh[200],ena[200],ek[200],eca[200];
double bestfitness[MAXGENES];
double max[MAXPARM];
double min[MAXPARM];
double maxfit=1e100;
char errfil[10], popfil[10],popfiltemp[10];

double fitness[MAXGENES];
int geneindex[MAXGENES];
int choice[MAXGENES*10];
FILE *fptr;
FILE *fptr2;
/*FILE *fptr3; */
double averagefitness,sumfitness;
int testx,testy;

void main(/*int argc,char *argv[]*/void)
{


          int x,loop,memhand,xcount,ycount;
          char input[10],gentab[10];
          char tab1;
```

```
        float proby;

    /*  if(argc<2)
         {
          printf("you must enter the command line again \n");
          exit(1);
          }    */

        /**** Memory allocation for large population arrays ***/
        for(memhand=1;memhand<=400+1;memhand++)
         {

          genepool[memhand] = (long int
far*)farmalloc((NUMPARM*(long)sizeof(long))+1);
          if(genepool[memhand] == NULL)
            {
                printf("OUT OF MEMORY \n");
                exit(1);
            }
          }

                /*  printf("enter the name of the error file \n");
                    scanf("%s",errfil);
                    printf("enter the popn. file \n");
                    scanf("%s",popfil);     */

        /*   printf(" first argument %s \n", argv[1]);
          printf("second argument %s \n", argv[2]);
          strcpy(popfil,*(argv+2));
          printf("popn. file %s\n",popfil);
           strcpy(popfiltemp,popfil);  */

           strcpy(errfil,"ca52.dat");
           printf("error file %s\n",errfil);

           printf("file handling \n");

        filhand();

     printf("Randomizing \n");

        setrand();

        numgenes = 400;/*** change this later ***/
        crosspool = 160 ;
        printf("filling the first gene\n");

        fillgene();
```

```
  printf("filling the pool \n");
fillpool();


  numcycles = 0;


  do
    {

      numcycles++;

      strcpy(popfil,popfiltemp);
      itoa(numcycles,gentab,10);
      strcat(popfil,gentab);
      printf("Cycle no. %d\n",numcycles);
      strcat(popfil,".dat");
      printf("popn. file %s\n",popfil);
   /*  if((fptr3=fopen(popfil,"w"))== NULL)
          {
            printf("sorry cannot open the popn. file\n");
            exit(1);
          }  */

      printf("Evaluating and Sorting....\n");

      evalgenes();
      sort();
  /*   for(xcount=1;xcount<=numgenes;xcount++)
       {
          printf("%d, %ld, %ld, %ld, %ld, %ld, %lf \n", xcount,
                genepool[geneindex[xcount]][1],
                genepool[geneindex[xcount]][2],
                genepool[geneindex[xcount]][3],
                genepool[geneindex[xcount]][4],
                genepool[geneindex[xcount]][5],
                fitness[geneindex[xcount]]);
       sleep(1);
       }

       sleep(5); */
     if((fitness[geneindex[1]]) > THRESHOLD)
          {

            printf("Breeding \n");
            rankscale();
            preselect();
            select();
```

```
            evalgenes();
        sort();
            proby = randnew();


            if(proby < CROSSRATE)
             {
                 printf("Crossing.... \n");
           evalgenes();
                 crossgenes();
             }

           printf("MUTATING GENES \n");
       evalgenes();
           mutategene();

          }
     evalgenes();
     sort();
     stat();
 /*  fclose(fptr3);   */
  }
while((numcycles<300) && (fitness[geneindex[1]]  > THRESHOLD));


/***** freeing the memory taken up by the genepopulation ****/
 for(memhand =1;memhand < 400+1; memhand ++)
       farfree(genepool[memhand]);
/*  printf("This machine now has %lu bytes free \n",farcoreleft()); */
}


/*********** file handling function ***********/
void filhand(void)
 {
  int tempcount;
  clrscr();
   numdata =0;
  if((fptr=fopen("ca5.txt","r"))== NULL)
       {
         printf("sorry cannot open this file\n");
         exit(1);
       }

while(fscanf(fptr,"%f%f%f%f%f%f%f%f",&enh[numdata],&ena[numdata],&ek[numd
ata],&eca[numdata],

&anh[numdata],&ana[numdata],&ak[numdata],&aca[numdata]) != EOF)
       {
```

```
        /*
printf("%f,%f,%f,%f\n",enh[numdata],ena[numdata],ek[numdata],eca[numdata]);

printf("%f,%f,%f,%f\n",anh[numdata],ana[numdata],ak[numdata],aca[numdata]); */
       numdata = numdata +1;
      }
        fclose(fptr);

   printf("NUMBER OF DATAPOINTS %d\n",numdata);
 /*  for(tempcount=0;tempcount<=39;tempcount++)
     {
     anh[tempcount] = anh[tempcount]*0.001;
     ana[tempcount] = ana[tempcount]*0.001;
     ak[tempcount] = ak[tempcount]*0.001;
     aca[tempcount] = aca[tempcount]*0.001;
   }   */
  }
/************* end of file handling function ******************/


/*************quicksorting procedure *********************/
 void sort(void)
  {
  int j,k;
  void qsortc(int l, int r);
  for(j=1;j<=numgenes;j++)
     geneindex[j] = j;
  for(k=1;k<=numgenes;k++)
     bestfitness[k] = fitness[k];
    qsortc(1,numgenes);
  }
 void qsortc(int l, int r)
 {
  int i,j;
  double x,y;
  int d_index;
  int pos;
  div_t cent;
  i=l;
  j=r;
  cent = div((l+r),2);
  pos = cent.quot;
  x = bestfitness[pos];

  do
   {
       while(bestfitness[i] < x && i< r) i++;
       while(x < bestfitness[j] && j> l) j--;
       if(i<=j)
```

```
        {
                y= bestfitness[i];
                bestfitness[i] = bestfitness[j]; /*** the values of the objective function
are sorted ***/
            bestfitness[j] = y;
                d_index= geneindex[i];          /*** but the genes themselves are not
sorted rather  ***/
                geneindex[i] = geneindex[j];   /*** indices are sorted ***/
                geneindex[j] = d_index;
                i= i+1;
                j= j-1;
        }

  }
  while(i<=j);
  if(l<j) qsortc(l,j);
  if(i<r) qsortc(i,r);


    }

 /************* end of quicksorting routine *****************/

 /***** Filling the first chromosome genes based on user input */
 /***** of initial guess of the parameters ****/
  void fillgene(void)
   {
    long int doubletoword(double value,int index);
    int counter;
    double valu[200];
    clrscr();
  /*   printf(" AMMONIUM ISE PARAMETER GUESSES \n\n");    */

        /*** AMMONIUM ISE PARAMETERS *****/

   /*     valu[1] =200;
          min[1] = 100;
          max[1] = 300;
          *(genepool[1]+1) = doubletoword(valu[1],1);

          valu[2] = 55;
          min[2] = 50;
          max[2] = 60;
         *(genepool[1]+2) = doubletoword(valu[2],2);

          valu[3] = 0.005;
          min[3] = 0.00001;
          max[3] = 0.01;
        *(genepool[1]+3) = doubletoword(valu[3],3);
```

```
  valu[4] = 0.5;
  min[4] = 0.001;
  max[4] = 1;
 *(genepool[1]+4) = doubletoword(valu[4],4);

valu[5] = 0.0001;
  min[5] = 0.00001;
  max[5] = 0.001;
 *(genepool[1]+5) = doubletoword(valu[5],5);  */


  /*** SODIUM ISE ****/
 /*  valu[6] = 500;
  min[6] = 100;
  max[6] = 1000;
 *(genepool[1]+6) = doubletoword(valu[6],6);

  valu[7] = 75;
  min[7] = 10;
  max[7] = 100;
 *(genepool[1]+7) = doubletoword(valu[7],7);

  valu[8] = 0.5;
  min[8] = 0.001;
  max[8] = 1;
 *(genepool[1]+8) = doubletoword(valu[8],8);

  valu[9] = 0.5;
  min[9] = 0.001;
  max[9] = 1;
 *(genepool[1]+9) = doubletoword(valu[9],9);

  valu[10] = 0.0001;
  min[10] = 0.00001;
  max[10] = 0.001;
 *(genepool[1]+10) = doubletoword(valu[10],10); */

 /*** POTASSIUM ISE ****/
/*    valu[11] = 500;
  min[11] = 100;
  max[11] = 1000;
 *(genepool[1]+11) = doubletoword(valu[11],11);

  valu[12] = 75;
  min[12] = 10;
  max[12] = 100;
  *(genepool[1]+12) = doubletoword(valu[12],12);
```

```
        valu[13] = 0.5;
        min[13] = 0.001;
        max[13] = 1;
      *(genepool[1]+13) = doubletoword(valu[13],13);

        valu[14] = 0.5;
        min[14] = 0.001;
        max[14] = 1;
       *(genepool[1]+14) = doubletoword(valu[14],14);

        valu[15] = 0.0001;
        min[15] = 0.00001;
        max[15] = 0.001;
       *(genepool[1]+15) = doubletoword(valu[15],15);  */

    /*** CALCIUM ISE ***/
        valu[1] = 200;
        min[1] = 100;
        max[1] = 300;
        *(genepool[1]+1) = doubletoword(valu[1],1);

        valu[2] = 25;
        min[2] = 20;
        max[2] = 30;
       *(genepool[1]+2) = doubletoword(valu[2],2);

        valu[3] = 0.0005;
        min[3] = 0.0000001;
        max[3] = 0.05;
       *(genepool[1]+3) = doubletoword(valu[3],3);

        valu[4] = 0.05;
        min[4] = 0.0000001;
        max[4] = 0.1;
       *(genepool[1]+4) = doubletoword(valu[4],4);

        valu[5] = 0.001;
        min[5] = 0.0001;
        max[5] = 0.5;
      *(genepool[1]+5) = doubletoword(valu[5],5);

/*   for(counter=1; counter<=5 ; counter++)
          {
           switch(counter)
            {
                case 1:
                printf("enter CELL POTENTIAL estimate");
                break;
                case 2:
```

```
                printf("enter SLOPE estimate");
                break;
                case 3:
                printf("enter Na+ SELECTIVITY COEFFICIENT estimate ");
                break;
                case 4:
                printf("enter K+ SELECTIVITY COEFFICIENT estimate ");
                break;
                case 5:
                printf("enter Ca2+ SELECTIVITY COEFFICIENT estimate ");
                break;
           }
          scanf("%lf",&valu[counter]);
          printf("\nenter LOWER BOUNDARY on the parameter");
          scanf("%lf",&min[counter]);
          printf("\nenter UPPER BOUNDARY on the parameter");
          scanf("%lf",&max[counter]);
        printf("\n\n"); */
          /* convert the double value of the first parameter */
          /* into a word and use it to encode the respective */
          /* gene on the  first chromosome       */



/*          genepool[1][counter]=doubletoword(valu[counter],counter);
         }

  clrscr();
  printf(" SODIUM ISE PARAMETER GUESSES \n\n");
  for(counter=6; counter<=10 ; counter++)
         {
          switch(counter)
           {
                case 6:
                printf("enter CELL POTENTIAL estimate");
                break;
                case 7:
                printf("enter SLOPE estimate");
                break;
                case 8:
                printf("enter NH4+ SELECTIVITY COEFFICIENT estimate ");
                break;
                case 9:
                printf("enter K+ SELECTIVITY COEFFICIENT estimate ");
                break;
                case 10:
                printf("enter Ca2+ SELECTIVITY COEFFICIENT estimate ");
                break;
           }
```

```
        scanf("%lf",&valu[counter]);
        printf("\nenter LOWER BOUNDARY on the parameter");
        scanf("%lf",&min[counter]);
        printf("\nenter UPPER BOUNDARY on the parameter");
        scanf("%lf",&max[counter]);
      printf("\n\n");      */
        /* convert the double value of the first parameter */
        /* into a word and use it to encode the respective */
        /* gene on the  first chromosome       */



     /*    genepool[1][counter]=doubletoword(valu[counter],counter);
       }
  clrscr();
  printf(" POTASSIUM ISE PARAMETER GUESSES \n\n");
  for(counter=11; counter<=15 ; counter++)
       {
        switch(counter)
         {
            case 11:
            printf("enter CELL POTENTIAL estimate");
            break;
            case 12:
            printf("enter SLOPE estimate");
            break;
            case 13:
            printf("enter NH4+ SELECTIVITY COEFFICIENT estimate ");
            break;
            case 14:
            printf("enter Na+ SELECTIVITY COEFFICIENT estimate ");
            break;
            case 15:
            printf("enter Ca2+ SELECTIVITY COEFFICIENT estimate ");
            break;
         }
        scanf("%lf",&valu[counter]);
        printf("\nenter LOWER BOUNDARY on the parameter");
        scanf("%lf",&min[counter]);
        printf("\nenter UPPER BOUNDARY on the parameter");
        scanf("%lf",&max[counter]);
      printf("\n\n");    */
        /* convert the double value of the first parameter */
        /* into a word and use it to encode the respective */
        /* gene on the  first chromosome       */


  /*       genepool[1][counter]=doubletoword(valu[counter],counter);
```

```
        }

    clrscr();
    printf(" CALCIUM ISE PARAMETER GUESSES \n\n");
    for(counter=16; counter<=20 ; counter++)
        {
          switch(counter)
           {
               case 16:
               printf("enter CELL POTENTIAL estimate");
               break;
               case 17:
               printf("enter SLOPE estimate");
               break;
               case 18:
               printf("enter NH4+ SELECTIVITY COEFFICIENT estimate ");
               break;
               case 19:
               printf("enter Na+ SELECTIVITY COEFFICIENT estimate ");
               break;
               case 20:
               printf("enter K+ SELECTIVITY COEFFICIENT estimate ");
               break;
           }
          scanf("%lf",&valu[counter]);
          printf("\nenter LOWER BOUNDARY on the parameter");
          scanf("%lf",&min[counter]);
          printf("\nenter UPPER BOUNDARY on the parameter");
          scanf("%lf",&max[counter]);
          printf("\n\n");     */

          /* convert the double value of the first parameter */
          /* into a word and use it to encode the respective */
          /* gene on the  first chromosome       */



      /*    genepool[1][counter]=doubletoword(valu[counter],counter);
        }   */


  }
/***************** end of filling the first chromosome **************/

/********* Filling the rest of the chromosomes in the gene pool******/
void fillpool(void)
 {
 int index, index2;
```

```
int temp;

double roundtempi,roundtempf;
for(index=2;index<=numgenes;index++)
   {
    printf("\n %d  ",index);

    for(index2=1;index2<=NUMPARM;index2++)
         {
               roundtempf= modf((double)(MAXWORD * randnew()),&roundtempi);
               if(roundtempf >0.5)
           roundtempi++;
           *(genepool[index]+index2) =(long int) roundtempi;

         }
       }

}
/************ end of function for filling the gene pool ***********/




   /********* GENE MUTATION OPERATOR FUNCTION **********/
   void mutategene(void)
    {
    /******* uses XOR to flip different bits ***********/
    /******* mutation rate is 1/population size  *******/

    /**** indices are used of which gene mutation is ***/
    /**** to occur at and also the chromosomes on    ***/
    /**** which it will occur  *************************/

    int mutepoint,atchrom,k,tempcount,tempcount2;
    double c,num =2.0;
    double roundfmut,roundfat,roundimut,roundiat;
    float proby,muterate;
    long int tempmeas;
    muterate = 1.0/numgenes;
  /*  fprintf(fptr3, "Mutating \n"); */


    for(k=1;k<=numgenes;k++)
         {

         proby = randnew();
         if(proby < muterate)
           {
```

```
            proby = randnew();


            roundfmut= modf((double)(proby*(WORDSIZE-1)),&roundimut);

            if(roundfmut>0.5)
            roundimut=roundimut+1;

            proby = randnew();
            roundfat= modf((double)(proby*(NUMPARM-1)),&roundiat);
            roundiat++;
            if(roundfat>0.5)
            roundiat=roundiat+1;
          /*  fprintf(fptr3,"chromosome %d mutated at gene %d at bitposition %d \n",
k, (int)roundiat,(int)roundimut);
            fprintf(fptr3,"i bp %ld val %lf
\n",*(genepool[k]+(int)roundiat),getchromosome(k,(int)roundiat));   */
           /******* PERFORM XOR OPERATION *****/
           c= pow(num,(double)roundimut);
          tempmeas= *(genepool[k] + (int)roundiat) ;
          tempmeas = tempmeas ^ (long int) c;
          *(genepool[k] +(int)roundiat) = tempmeas;
 /*       fprintf(fptr3,"a bp %ld val %lf
\n",*(genepool[k]+(int)roundiat),getchromosome(k,(int)roundiat));   */
        }
         }

    for(tempcount=1;tempcount<=numgenes;tempcount++)
            for(tempcount2 =1;tempcount2<=NUMPARM;tempcount2++)
              {
                 /* if(*(genepool[tempcount]+tempcount2) <= 0)
                    {
                printf("after mutation \n");
                     printf("chromosome %d gene %d value %ld\n",tempcount,
tempcount2,*(genepool[tempcount]+tempcount2));
                    } */
            }

   }

   /************END OF FUNCTION *******************/

   /***********CROSSOVER OPERATION FUNCTION **********/
   void crossgenes(void)
   {
   /***********************************************************/
   /*                    CROSSOVER OPERATOR                  */
   /*    Single point crossover using indices of both the gene at */
```

```
/*  which crossover is supposed to occur and also the bit     */
/*  number within the gene at which crossover occurs.         */
/*                                                  */
/*    Variables :                                              */
/*      insertindex,crossindex : refer to the points in the   */
/*                        chromosome popn. at which the */
/*                                crossed genes will be        */
/*                      inserted                  */
/*                                                 */
/*   template1[16],template2[16],temp1[16],temp2[16]          */
/*                        : refer to strings of the       */
/*                                binary character values of   */
/*                        a gene value               */
/*                                                 */
/*    crosspoint : bitnumber for crossover from the start     */
/*          of the chromosome                         */
/*                                                 */
/*    lastchrom  : the gene preceeding the gene at which      */
/*              crossover occurs                      */
/*****************************************************************/
void cross(int cpoint,long int gene1,long int gene2, long int *temp1,long int
*temp2);
div_t lastchromtemp;
int insertindex,crossindex,j,k,count;
long int template1[16],template2[16];
long int temp1,temp2;
int lastchrom,memhand;
int mate1,mate2,mateind,flag,countind;
int m1[MAXGENES],m2[MAXGENES];

long int far *temppool[402];
double round,roundtemp,proby,crosspoint;
insertindex = numgenes - (2 * crosspool);

/*** MEMORY ALLOCATION FOR THE TEMPORARY POPULATION ***/
 for(memhand=1;memhand <=400+1;memhand++)
     {
      temppool[memhand] = (long int
far*)farmalloc((NUMPARM*(long)sizeof(long))+1);
      if(temppool[memhand] == NULL)
        {
         printf("OUT OF MEMORY \n");
         exit(1);
        }
     }

/*   printf("This machine now has %lu bytes free \n",farcoreleft()); */

/* filling the temporary pool with the best chromosomes in the */
```

```
/* gene pool till the insertion point                    */

for(j=1;j<=numgenes;j++)
 {
  for(count=1;count<=NUMPARM;count++)
      *(temppool[j]+count) = *(genepool[geneindex[j]]+count);
 }

/*** setting up for looping through the gene pool ***/
/*   fprintf(fptr3,"Crossing \n");   */

 crossindex = -1;
insertindex -=2;

mateind =1;
do
{

    crossindex +=2;
  insertindex +=2;

    /**** Which bit position on the whole chromosome is this to ****/
  /**** happen at CROSSPOINT ****/
    proby = (double)randnew();

    round = proby * (NUMPARM * WORDSIZE-1);
    round = round +1;
  roundtemp = modf(round,&crosspoint);
    if(roundtemp > 0.5)
     crosspoint = crosspoint +1;
    if(crosspoint ==0)
      crosspoint =1;
    lastchromtemp = div(crosspoint,WORDSIZE);
  lastchrom =lastchromtemp.quot;
    /*if(lastchrom==0)
      lastchrom =1; */
    crosspoint = lastchromtemp.rem;
 /* printf("bit position cross point %lf\n",crosspoint);
    printf("gene %d\n",lastchrom);
    printf("chromosome 1: %d\n",crossindex);
    printf("chromosome 2: %d\n",crossindex+1);
    getch();  */

 /* copy the top 2 chromosomes into the first 2 chromosomes */
 /* beneath the insert index in the temppool.           */

 /**** STAGE 1 COPY GENE SEGMENTS TO THE LAST CHROMOSOME
*/
 /**** INTO THE TEMPPOOL CHROMOSOMES DIRECTLY              */
```

```
            mate1 = randnew()*(numgenes-1)+1;
            m1[mateind] = mate1;

            do{
              flag =0;
              mate2 = randnew()*(numgenes-1)+1
                          ;
              for(countind =1;countind<=(mateind-1);countind++)
                  {
                   if((mate1==m1[countind])&&(mate2==m2[countind]))
                      flag =1;
                  }
           }
            while((mate1==mate2)||(flag ==1));

            m2[mateind] =mate2;
            mateind++;

     /*     fprintf(fptr3,"mate 1 %d\n",mate1);
            for(j=1;j<=NUMPARM;j++)
              {
              fprintf(fptr3,"%ld, ",*(temppool[mate1]+j));
              fprintf(fptr3,"%lf\n" ,getchromosome(mate1,j));
              }


            fprintf(fptr3,"mate 2 %d\n",mate2);
            for(j=1;j<=NUMPARM;j++)
              {
                  fprintf(fptr3,"%ld, ",*(temppool[mate2]+j));
                  fprintf(fptr3,"%lf\n" ,getchromosome(mate2,j));
              }
           fprintf(fptr3,"cross at gene %d  bit %d \n", lastchrom+1,(int)crosspoint);    */

            if(lastchrom >0)
            {
            for(j=1;j<=lastchrom;j++)
              {
               *(temppool[insertindex]+j) = *(genepool[geneindex[mate1]]+j);
               *(temppool[insertindex+1]+j) =*(genepool[geneindex[mate2]]+j);
              }
             }
     /*** AFTER THE LAST CHROMOSOME SKIP THE BROKEN ONE        */
     /*** SWAP THE GENE SEQUENCES OF THE RELEVANT
CHROMOSOMES  */
          for(j=lastchrom+2;j<=NUMPARM;j++)
          {
```

```
             *(temppool[insertindex]+j) = *(genepool[geneindex[mate2]]+j);
            *(temppool[insertindex+1]+j) =*(genepool[geneindex[mate1]]+j);
           }
          cross(crosspoint,*(genepool[geneindex[mate1]]+lastchrom
+1),*(genepool[geneindex[mate2]]+lastchrom+1),&temp1,&temp2);

        *(temppool[insertindex]+lastchrom+1)=temp1;
        *(temppool[insertindex+1]+lastchrom+1) =temp2;
     /* fprintf(fptr3,"after crossing \n");
        fprintf(fptr3,"mate1 \n");
        for(j=1;j<=NUMPARM;j++)
          {
              fprintf(fptr3,"%ld, ",*(temppool[insertindex]+j));
              fprintf(fptr3,"%lf\n" ,getchromosome(insertindex,j));
          }

      fprintf(fptr3,"\n mate 2 \n");
        for(j=1;j<=NUMPARM;j++)
          {
              fprintf(fptr3,"%ld, ",*(temppool[insertindex+1]+j));
              fprintf(fptr3,"%lf\n" ,getchromosome(insertindex+1,j));
          }   */


        }


       while(crossindex < (2*crosspool)-2);

        for(j=1;j<=numgenes+1;j++)
         {
           for(count=1;count<=NUMPARM;count++)
          *(genepool[j]+count) = *(temppool[j]+count);
         }
       /*** FREEING MEMORY FROM THE TEMPORARY POPULATION
****/
       for(memhand=1;memhand <=400+1;memhand++)
        farfree(temppool[memhand]);

   }
   /*** end of function **/
   /******************************************************************/
   /*                     GENE CROSSING                            */
   /*    Crossing the bits of the respective genes by converting the */
   /*  binary representation of the denary number into a binary   */
   /*  array (with a bit mask) and performing the sawp at the      */
   /*  bit position on the gene. The resultant genes are         */
   /*  reconverted to denary numbers which are returned to the     */
   /*  calling function by reference via temp1 and temp2.        */
```

```
/*****************************************************************/

void cross(int cpoint,long int gene1,long int gene2,long int *temp1,long int *temp2)
 {
 int tempgen1[16],tempgen2[16];
 int gene1st[16],gene2st[16];
 int count;
 double num,conv1,conv2;
 unsigned int mask = 0x8000;

 /**** convert number from 0 - 2^16 to its binary string ****/
 for(count=0;count<16;count++)
   {
    gene1st[15 - count] = (mask & gene1) ? 1 : 0;
    gene2st[15 - count] = (mask & gene2) ? 1 : 0;
    mask >>=1;
  }
 /***** copy the binary components of the gene into the ***/
 /***** template genes up to the crossover  point      ***/

 for(count=0;count<cpoint;count++)
  {
   tempgen1[count] = gene1st[count];
   tempgen2[count] = gene2st[count];
    }

 /**** swap the base pairs at the cross point ***/
   for(count=cpoint;count<16;count ++)
     {
      tempgen1[count] = gene2st[count];
      tempgen2[count] = gene1st[count];
   }

 /***** reconvert the binary strings to their denary  ***/
 /**** equivalent between the range 0 - 2^16         ***/

   conv1=0;
   conv2=0;
   num = 2.0;
   for(count=0;count<16;count ++)
    {
      conv1 = conv1 + tempgen1[count]*pow(num,(double)count);
      conv2 = conv2 + tempgen2[count]*pow(num,(double)count);
   }
   *temp1 = (long int) conv1;
   *temp2 = (long int) conv2;

 }
 /**** end of gene crossing function *****************/
```

```
/****************************************************/
/* Converting the user input double value to a long   */
/* in the range 0 - 2^16 (word)                  */
/****************************************************/
long int doubletoword(double value,int index)
 {
  double wordtemp,word;
    double c;
    wordtemp = (MAXWORD*(value - min[index]))/(max[index] -min[index]);
    word = modf(wordtemp,&c);
    if(word > 0.5)
      c++;
    return((long int)c);
    }
    /****** end of double to word function **********/

/**********************************************************/
/*  EVALUATING THE FITNESS OF THE CHROMOSOMES IN THE        */
/*  POPULATION                                             */
/*  and updating the average fitness of all the chromosomes */
/*  in the population.                          */
/**********************************************************/

void evalgenes(void)
     {
     int c,tempcount,tempcount2;
   double calcfunc(int c);
     averagefitness = 0.0;

   for(tempcount=1;tempcount<=numgenes;tempcount++)
        for(tempcount2 =1;tempcount2<=NUMPARM;tempcount2++)
         {
             if(*(genepool[tempcount]+tempcount2) < 0)
               {
                printf("chromosome %d gene %d value %ld\n",tempcount,
tempcount2,*(genepool[tempcount]+tempcount2));
            exit(0);
               }
       }

     for(c=1;c<=numgenes;c++)
       {

        fitness[c] = calcfunc(c);
        averagefitness = averagefitness + fitness[c];
       }
       averagefitness = averagefitness/numgenes;
  /*   printf("0-100 \n");
```

```
          for(c=100;c<=200;c++)
          {
           printf("gene %d \n",c);
           fitness[c] = calcfunc(c);
           averagefitness = averagefitness + fitness[c];
          }
          averagefitness = averagefitness/100;
          printf("100-200\n");      */
        }
/****** END OF EVALUATION FUNCTION *****************************/

/********** DETERMINATION OF MSE FOR A GIVEN CHROMOSOME
******/
        double calcfunc(int c)
         {
          double getchromosome(int geneindex,int chromoindex);
          double errs,ecellnh,ecellna,ecellk,ecellca;
          double snh,sna,sk,sca;
          double knhna,knhk,knhca;
          double knanh,knak,knaca;
          double kknh,kkna,kkca;
          double kcanh,kcana,kcak;
          double ycalctemp,ycalc;
          double emodnh,emodna,emodk,emodca;
          int j,counter;


        /*  ecellnh = getchromosome(c,1);
          snh = getchromosome(c,2);
          knhna = getchromosome(c,3);
          knhk = getchromosome(c,4);
          knhca = getchromosome(c,5);
          printf(" ecellnh %lf\n",ecellnh);
          printf("snh %lf\n",snh);
          printf("knhna %lf\n",knhna);
          printf("knhk %lf\n",knhk);
          printf("knhca %lf\n",knhca);
          printf("\n");

       ecellna = getchromosome(c,1);
          sna = getchromosome(c,2);
          knanh = getchromosome(c,3);
          knak = getchromosome(c,4);
          knaca = getchromosome(c,5);

      printf(" ecellna %lf\n",ecellna);
          printf("sna %lf\n",sna);
          printf("knanh %lf\n",knanh);
          printf("knak %lf\n",knak);
```

```
        printf("knaca %lf\n",knaca);
        printf("\n");

      ecellk = getchromosome(c,11);
        sk = getchromosome(c,12);
        kknh = getchromosome(c,13);
        kkna = getchromosome(c,14);
        kkca = getchromosome(c,15);
      printf(" ecellk %lf\n",ecellk);
        printf("sk %lf\n",sk);
        printf("kknh %lf\n",kknh);
        printf("kkna %lf\n",kkna);
        printf("kkca %lf\n".kkca);
        printf("\n");     */

      ecellca = getchromosome(c,1);
        sca = getchromosome(c,2);
        kcanh = getchromosome(c,3);
        kcana = getchromosome(c,4);
        kcak = getchromosome(c,5);
    /*  printf(" ecellca %lf\n",ecellca);
        printf("sca %lf\n",sca);
        printf("kcanh %lf\n",kcanh);
        printf("kcana %lf\n",kcana);
        printf("kcak %lf\n",kcak);
        printf("\n");*/

       /*  printf("calculating the model parameters \n\n");

      clrscr(); */
        errs=0;
        counter =-1;

        do
         {
         counter++;
       /*    emodnh = anh[counter] + ana[counter]*knhna + ak[counter]*knhk +
                     aca[counter]*knhca;

         if(emodna ==0)
              {
               printf(" ZERO ERROR ON THE LOG CHECK THIS OUT FOR
CHROMOSOME %d\n",c);
               exit(0);
              }

       emodna = ana[counter] + anh[counter]*knanh + ak[counter]*knak +
                     aca[counter]*knaca;
```

```
        if(emodna ==0)
             {
              printf(" ZERO ERROR ON THE LOG CHECK THIS OUT FOR
CHROMOSOME %d\n",c);
              exit(0);
             }


        emodk = ak[counter] + anh[counter]*kknh + ana[counter]*kkna +
                  aca[counter]*kkca;

        if(emodk ==0)
             {
              printf(" ZERO ERROR ON THE LOG CHECK THIS OUT FOR
CHROMOSOME %d\n",c);
              exit(0);
             }


          printf(" K model %lf \n",emodk);  */

          emodca = aca[counter] + ana[counter]*kcana +
               anh[counter]*kcanh + ak[counter]*kcak ;

         if(emodca ==0)
             {
              printf(" ZERO ERROR ON THE LOG CHECK THIS OUT FOR
CHROMOSOME %d\n",c);
              exit(0);
             }

        /*   printf(" Na model %lf \n",emodca);
          getch();   */

              /*  emodnh = snh*log10(emodnh) + ecellnh;
         emodna = sna*log10(emodna) + ecellna;
                   emodk  = sk*log10(emodk) + ecellk; */
            emodca = sca*log10(emodca) + ecellca;

        errs = errs + ((emodca - eca[counter])*(emodca -
eca[counter])/(eca[counter]*eca[counter]));

                       /*  +(emodna - ena[counter])*(emodna - ena[counter]) +
                         (emodk - ek[counter])*(emodk - ek[counter]) +
                         (emodca - eca[counter])*(emodca - eca[counter]) ;
                     printf("counter %d\n",counter);
                     getch();
                   clrscr();*/
        }
```

```
        while((counter < numdata-1) && (errs <=maxfit));
      errs = sqrt(errs);
      return(errs);
        }
 /****** end of MSE determination ***********/
 /****** Convert gene word to a decodes value  ****/
 double getchromosome(int geneindex,int chromoindex)
         {
         double s;
         s = (double)*(genepool[geneindex]+chromoindex)/(double)MAXWORD;
       s = s* (max[chromoindex] - min[chromoindex]) + min[chromoindex];
       return(s);
       }
/****** end of conversion function *******/
void stat(void)
{
      void sort(void);
      void qsortc(int l,int r);
      void evalgenes(void);
      double getchromosome(int geneindex,int chromindex);
      double dump[MAXPARM];
      int dumpcount[MAXGENES],hitlist[MAXGENES],tempcount;
    int inc,chk,arr,l,m,n,counter,dumpchk,arrchk,loop;
      int pop,tempcount2;
      div_t filtemp;

    pop =1;
      l = 1;
      inc = 1;
      chk = 0;
      /*       printf("checking copies of a chromosome \n");  */
      for(tempcount=1;tempcount<=numgenes;tempcount++)
         for(tempcount2 =1;tempcount2<=NUMPARM;tempcount2++)
          {
          /*  if(*(genepool[tempcount]+tempcount2) <= 0)
               {
                printf("chromosome %d gene %d value %ld\n",tempcount,
tempcount2,*(genepool[tempcount]+tempcount2));
               }  */
        }
      do
        {
         dumpcount[l] = 1;
             for(m=1;m<=NUMPARM;m++)
                     dump[m] = *(genepool[geneindex[l]]+m);
             dumpcount[l]=1;
             for(n=l+1;n<=numgenes;n++)
               {
             counter = 0;
```

```
                for(arrchk=1;arrchk<=NUMPARM;arrchk++)
                {
                        if(*(genepool[geneindex[n]]+arrchk) == dump[arrchk])
                                counter ++;
                  }
                if(counter >= NUMPARM)
                  {
                        dumpcount[l] = dumpcount[l] +1;
                        hitlist[inc] = n;
                        inc++;
                  }
          }

        l++;
        do
          {
   chk =1;
            for(dumpchk =1;dumpchk<inc;dumpchk++)
              {
                if(l==hitlist[dumpchk])
                  {
                        l++;
                        chk=0;
                  }
              }
          } while(chk ==0);
}
      while(l<=numgenes);
      /******          REPORT GENERATION      *******/



      printf("Best fit: %lf\n", fitness[geneindex[1]]);
      printf("average fitness %lf\n",averagefitness);

      printf("AMMONIUM ISE\n");
      printf("Ecell %lf\n",getchromosome(geneindex[1],1));
      printf("slope %lf\n",getchromosome(geneindex[1],2));
      printf("Na+ selectivity coefficient %lf\n",getchromosome(geneindex[1],3));
      printf("K+ selectivity coefficient %lf\n",getchromosome(geneindex[1],4));
      printf("Ca2+ selectivity coefficient %lf\n",getchromosome(geneindex[1],5));

      if((fptr2=fopen(errfil,"a"))== NULL)
            {
              printf("sorry cannot open the error file\n");
              exit(1);
            }

                        /*filtemp = div(numcycles,5);
```

```
                                if(filtemp.rem == 0)
                                {  */

        fprintf(fptr2,"%lf %lf %lf %lf %lf %lf %lf\n",
                fitness[geneindex[1]],averagefitness,getchromosome(geneindex[1],1),
                getchromosome(geneindex[1],2),getchromosome(geneindex[1],3),
                getchromosome(geneindex[1],4),getchromosome(geneindex[1],5));
        fprintf(fptr2,"\n\n");

                            /*    }    */

                            /*  printf(" ** POPULATION STATISTICS ***\n");  */
        pop =1;

      do
        {
        printf(" Chromosome %d, copies %d\n",geneindex[pop],dumpcount[pop]);
   /*   printf("cell pot %lf\n",getchromosome(geneindex[pop],1));
        printf("slope %lf\n",getchromosome(geneindex[pop],2));
        printf("na sel coeff %lf\n",getchromosome(geneindex[pop],3));
        printf("k sel coeff %lf\n",getchromosome(geneindex[pop],4));
        printf("ca sel coeff %lf\n",getchromosome(geneindex[pop],5));  */

        printf("fitness %lf\n",fitness[geneindex[pop]]);

                                /*  if(filtemp.rem == 0)
                                    {  */

  /*    fprintf(fptr3,"final population \n");
        fprintf(fptr3,"%d %d %lf \n",geneindex[pop],dumpcount[pop],
                        fitness[geneindex[pop]]);
        fprintf(fptr3,"ecell %lf bitvalue %ld
\n",getchromosome(geneindex[pop],1),*(genepool[geneindex[pop]]+1));
fprintf(fptr3,"slope %lf bitvalue %ld
\n",getchromosome(geneindex[pop],2),*(genepool[geneindex[pop]]+2));
fprintf(fptr3,"kna %lf bitvalue %ld
\n",getchromosome(geneindex[pop],3),*(genepool[geneindex[pop]]+3));
fprintf(fptr3,"kk %lf bitvalue %ld
\n",getchromosome(geneindex[pop],4),*(genepool[geneindex[pop]]+4));
fprintf(fptr3,"kca %lf bitvalue %ld
\n",getchromosome(geneindex[pop],5),*(genepool[geneindex[pop]]+5));


        fprintf(fptr3,"\n\n");  */

                                /*   } */
        fclose(fptr2);

        pop++;
```

```
        do
         {
                chk =1;
                for(dumpchk =1;dumpchk < inc;dumpchk ++)
                 {
                 if(pop == hitlist[dumpchk])
                  {
                   pop ++;
                   chk =0;
                   }
                 }
         }
       while(chk ==0);
  }while(pop <=numgenes);

/* getch(); */
 }

 void prescale(void)
  {
   int count;
   double m, c;
   double alpha;
   alpha = 1.5;
   m = (1.0 - (1.0/alpha))*averagefitness/(averagefitness-fitness[geneindex[1]]);
   c = averagefitness-m*averagefitness;
  for(count=1;count<=numgenes;count++)
    fitness[geneindex[count]] = fitness[geneindex[count]]*m + c;
  }

 void rankscale(void)
  {
   int count;
   double m,c;
   double alpha;
   alpha = 1.1;
   m = 2.0*(alpha-1)/numgenes;
   c= 2.0 - alpha;
   sumfitness = 0.0;
   for(count=1;count <=numgenes;count++)
    {
        fitness[geneindex[count]] = (numgenes-count+1)*m + c;
        sumfitness = sumfitness + fitness[geneindex[count]];
    }
  }
```

```
void preselect(void)
 {
  int j,k,jassign,winner;
  double fraction[MAXGENES];
  double expected;

  j=0;
  k=0;
  do
      {
       j=j+1;
       expected = fitness[geneindex[j]];
       jassign = (int)floor(expected);
       fraction[j] = expected - jassign;


       while(jassign >0)
         {
          k = k+1;
          jassign = jassign -1;
          choice[k] = j;
         }

      }
     while(j<numgenes);

     j=0;



     while(k < numgenes)
       {
       j= j+1;
       if(j>numgenes)
          j=1;
       if(fraction[j] >0.0)
          {
          if(flip(fraction[j]) ==1)
                 winner =1;
          else winner =0;

          if(winner ==1)
                {
                k= k+1;
                choice[k] =j;
                fraction[j] = fraction[j] -1.0;
                }
          }
```

```
        }
      }




void select(void)
  {
   int jpick,count,memhand;
   int tempind,breednum;
   int nremain;
   long int far *tempool[402];
   int k,l,hold;


        printf("This machine now has %lu bytes free \n",farcoreleft());
        /*** memory allocation for temporary population array ****/
         for(memhand=1;memhand<=400+1;memhand++)
          {
           tempool[memhand] =(long int
far*)farmalloc((NUMPARM*sizeof(long))+1);
           if(tempool[memhand] == NULL)
                {
                 printf("OUT OF MEMORY \n");
                 exit(1);
                }
          }



   nremain = numgenes;
   breednum =1;
   do
   {
    jpick = pickrand(1,nremain);

    for(count=1;count<=NUMPARM;count++)
          tempool[breednum][count] = genepool[geneindex[choice[jpick]]][count];

    hold = choice[jpick];
    choice[jpick] = choice[nremain];
    choice[nremain] = hold;
    nremain = nremain -1;
    breednum = breednum +1;
    }
   while(breednum <= numgenes);
```

```
for(l=1;l<=numgenes+1;l++)
      {
       for(count =1;count<=NUMPARM;count++)
            *(genepool[l]+count) = *(tempool[l]+count);

       }


  /**** freeing the memory held by the temporary population ***/

     for(memhand = 1;memhand <= 400+1;memhand++)
        farfree(tempool[memhand]);

/*    printf("This machine now has %lu bytes free \n",farcoreleft());  */




}
```

Simplex.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NMAX 5000
#define ALPHA 1.0
#define BETA  0.5
#define GAMMA 2.0
#define MAXDIM 10

double amotry(float fac);
void filhand(void);
void fillrest(void);
void fillvertex(void);
void evalvertices(void);
double calcfunc(double *c);
double max[MAXDIM],min[MAXDIM];

float anh[100], ana[100],ak[100],aca[100];
float enh[100],ena[100],ek[100],eca[100];
double p[MAXDIM +1][MAXDIM],y[MAXDIM];
double fitness[MAXDIM],averagefitness;
double psum[MAXDIM],ptry[MAXDIM];
int numdata,ndim, mpts;
int ilo,ihi,inhi;
void main(void)
 {
 int j,nfunk;
 int i;
 double ytry,ysave,sum,rtol;
FILE *fptr;
 double temp[5];
 char name[10];


 printf("enter the desired name of the error file \n");
 scanf("%s",name);
 if((fptr= fopen(name,"w")) == NULL)
  {
   printf("sorry cannot open statistics log \n");
   exit(1);
  }
 randomize();

 mpts = 6;
 ndim = 5;
  nfunk =0;
  filhand();

 /*fillvertex();
```

```
fillrest(); */


p[1][1]= 274.981308;
p[1][2] = 53.125048;
p[1][3] =0.164889;
p[1][4] =5e-4;
p[1][5] =0.004777;

p[2][1] =266.405737;
p[2][2] =50.468910;
p[2][3] =0.148913;
p[2][4] =5e-4;
p[2][5] = 0.004207;

p[3][1] = 276.031128;
p[3][2] = 53.75021;
p[3][3] = 0.187861;
p[3][4] = 5.0e-6;
p[3][5] = 0.005489;

p[4][1] = 268.419928;
p[4][2] = 50.742046;
p[4][3] = 0.133174;
p[4][4] = 5.0e-6;
p[4][5] = 0.00507;

p[5][1] =272.16907;
p[5][2] =52.128939;
p[5][3] =0.145842;
p[5][4] =5.0e-6;
p[5][5] = 0.004778;

p[6][1] = 270.308995;
p[6][2] = 51.504234;
p[6][3] = 0.148799;
p[6][4] = 5.0e-6;
p[6][5] = 0.0047777;


do{
        for(i=1;i<=mpts;i++)
          y[i] =calcfunc(*(p+i));


        printf("help \n");
        printf("help again \n");

        /*** sorting the vertices ***/
```

```
ilo =1;

if(y[1] > y[2])
  {
   ihi =1;
   inhi =2;
  }
else
  {
   ihi=2;
   inhi =1;
  }


for(i=1;i<=mpts;i++)
  {
   if(y[i] < y[ilo])
          ilo = i;
   if(y[i]> y[ihi])
     {
          inhi = ihi;
          ihi = i;
     }
   else if(y[i] > y[inhi])
          if(i!= ihi)
             inhi= i;
   }

for(j=1;j<=ndim;j++)
     {
      sum=0;
      for(i=1;i<=mpts;i++)
             sum = sum+p[i][j];
      psum[j] = sum ;
     }

if(nfunk >= NMAX)
     {
       printf("maximum number of iterations exceeded \n");
       exit(0);
     }



     /*** initail extrapolation by alpha ***/
ytry = amotry(-ALPHA);
printf("help \n");
printf("help again \n");

/*** if the reflected point gives a better result than the best point**/
```

```
 /*** try an additional expansion by gamma                    **/

  if(ytry<=y[ilo])
      ytry= amotry(GAMMA);
  else if(ytry >= y[inhi]) /*** if it is worse than the second highest **/
       {
        ysave=y[inhi];  /** contract to find a medium point        **/
        ytry= amotry(BETA);
        if(ytry >=ysave)
         {
          for(i=1;i<=mpts;i++)
           {
              if(i!=ilo)
               {
                for(j=1;j<=ndim;j++)
                 {
                  psum[j] = 0.5*(p[i][j] + p[ilo][j]);
                  p[i][j] = psum[j];
                 }
                y[i] = calcfunc(psum);
               } /*** matching if(i!ilo) **/
           } /*** matching for i=1 to mpts ***/

          for(j=1;j<=ndim;j++)
           {
              sum = 0.0;
              for(i=1;i<=mpts;i++)
                 sum = sum +p[i][j];
              psum[j] = sum;
           }
       } /*** matches if(ytry >=ysave) **/
    } /** matches if(ytry >= y[inhi]) **/
nfunk++;
evalvertices();
fprintf(fptr,"%d %lf %lf %lf %lf %lf %lf %lf \n",nfunk, averagefitness,
                 fitness[ilo],p[ilo][1],p[ilo][2],p[ilo][3],p[ilo][4],
                 p[ilo][5]);
printf("cycle %d \n",nfunk);
printf(" cell potn. %lf\n", p[ilo][1]);
printf("slope %lf\n",p[ilo][2]);
printf("sodium selectivity coefficient %lf\n",p[ilo][3]);
printf("potassium selectivity coefficient %lf\n",p[ilo][4]);

printf("calcium selectivity coefficient %lf\n",p[ilo][5]);




}
while(nfunk <=600);
```

```
fclose(fptr);

}



double calcfunc(double *c)
{
        double errs,ecellnh,ecellna,ecellk,ecellca;
        double snh,sna,sk,sca;
        double knhna,knhk,knhca;
        double knanh,knak,knaca;
        double kknh,kkna,kkca;
        double kcanh,kcana,kcak;
        double ycalctemp,ycalc;
        double emodnh,emodna,emodk,emodca;
        int j,counter;


        ecellk = *(c+1);
        sk = *(c+2);
        kknh = *(c+3);
        kkna = *(c+4);
        kkca = *(c+5);
                        /*  printf(" ecellnh %lf\n",ecellnh);
                             printf("snh %lf\n",snh);
                             printf("knhna %lf\n",knhna);
                             printf("knhk %lf\n",knhk);
                           printf("knhca %lf\n",knhca);
                             printf("\n");

                    ecellna = getchromosome(c,6);
                             sna = getchromosome(c,7);
                             knanh = getchromosome(c,8);
                             knak = getchromosome(c,9);
                             knaca = getchromosome(c,10);
                             printf(" ecellna %lf\n",ecellna);
                             printf("sna %lf\n",sna);
                             printf("knanh %lf\n",knanh);
                             printf("knak %lf\n",knak);
                             printf("knaca %lf\n",knaca);
                             printf("\n");

                    ecellk = getchromosome(c,11);
                             sk = getchromosome(c,12);
                             kknh = getchromosome(c,13);
                             kkna = getchromosome(c,14);
                             kkca = getchromosome(c,15);
                             printf(" ecellk %lf\n",ecellk);
```

```
printf("sk %lf\n",sk);
printf("kknh %lf\n",kknh);
printf("kkna %lf\n",kkna);
printf("kkca %lf\n",kkca);
printf("\n");

ecellca = getchromosome(c,16);
sca = getchromosome(c,17);
kcanh = getchromosome(c,18);
kcana = getchromosome(c,19);
kcak = getchromosome(c,20);
printf(" ecellca %lf\n",ecellca);
printf("sca %lf\n",sca);
printf("kcanh %lf\n",kcanh);
printf("kcana %lf\n",kcana);
printf("kcak %lf\n",kcak);
printf("\n");*/

/*  printf("calculating the model parameters \n\n");
clrscr(); */
errs=0;
counter =-1;

do
 {
 counter++;
  /*  emodnh = anh[counter] + ana[counter]*knhna + ak[counter]*knhk +
        aca[counter]*knhca;

        if(emodnh ==0)
         {
         printf(" ZERO ERROR ON THE LOG CHECK THIS OUT
FOR CHROMOSOME %d\n",c);
         exit(0);
        }

        emodna = ana[counter] + anh[counter]*knanh +
ak[counter]*knak +
          aca[counter]*knaca;

      if(emodnh ==0)
         {
         printf(" ZERO ERROR ON THE LOG CHECK THIS OUT
FOR CHROMOSOME %d\n",c);
         exit(0);
         }  */

emodk = ak[counter] + anh[counter]*kknh + ana[counter]*kkna +
      aca[counter]*kkca;
```

```
                    /*  printf(" K model %lf \n",emodk);

                         emodca = aca[counter] + ana[counter]*kcana +
                                anh[counter]*kcanh + ak[counter]*kcak ;

                      printf(" Na model %lf \n",emodca);
                      getch();

                              emodnh = snh*log10(emodnh) + ecellnh;
                              emodna = sna*log10(emodna) + ecellna; */
                      emodk  = sk*log10(emodk) + ecellk;
                                /* emodca = sca*log10(emodca) + ecellca;

                            errs = errs + (emodnh - enh[counter])*(emodnh -
enh[counter])/(enh[counter]*enh[counter]);
                              errs = errs +   +(emodna - ena[counter])*(emodna -
ena[counter])/(ena[counter]*ena[counter]);*/
                         errs = errs +  (emodk - ek[counter])*(emodk -
ek[counter])/(ek[counter]*ek[counter]);
                               /*    errs = errs + (emodca - eca[counter])*(emodca -
eca[counter])/(eca[counter]*eca[counter]) ;

                      printf("counter %d\n",counter);
                      getch();
                    clrscr();*/
            }
            while(counter < (numdata-1));
          errs = sqrt(errs);
            return((double)errs);
}
/*** END OF FUNCTION EVALUATION ****/

 void fillvertex(void)
   {
    int counter;
    double valu[200];
    clrscr();
/*****************************************************************/
/** sets up the first vertex according to initial guess and asks for boundaries  **/
/** for the parameters for the other vertices.                                   **/
/*****************************************************************/
       /*** AMMONIUM ISE PARAMETERS *****/
         for(counter=1; counter<=5 ; counter++)
               {
                switch(counter)
                  {
                     case 1:
                     printf("enter CELL POTENTIAL estimate");
                     break;
                     case 2:
```

```
                    printf("enter SLOPE estimate");
                    break;
                    case 3:
                    printf("enter Na+ SELECTIVITY COEFFICIENT estimate ");
                    break;
                    case 4:
                    printf("enter K+ SELECTIVITY COEFFICIENT estimate ");
                    break;
                    case 5:
                    printf("enter Ca2+ SELECTIVITY COEFFICIENT estimate ");
                    break;
                   }
                    scanf("%lf",&valu[counter]);
                    printf("\nenter LOWER BOUNDARY on the parameter");
                    scanf("%lf",&min[counter]);
                    printf("\nenter UPPER BOUNDARY on the parameter");
                    scanf("%lf",&max[counter]);
                    printf("\n\n");

                    p[1][counter]=valu[counter];
          }


  }
/***************** end of filling the first VERTEX *************/


/********* Filling the rest of the VERTICES in the SIMPLEX ******/
void fillrest(void)
 {
 int index, index2,tab,inpow;
 int temp,guess2;
 double range,guess,fill2;
 double roundtempi,roundtempf;
 double fill;
 for(index=2;index<=mpts;index++)
    {
     for(index2=1;index2<=ndim;index2++)
         {
              range = max[index2] - min[index2];
              guess = log10(range);
              guess2 = floor(guess);
              range = range/pow10(guess2);
              if(guess2 >=0)
               {
               guess2=guess2+3.0;
               tab=1;
               }
               else
                {
```

```
                inpow = abs(guess2)+3;
                guess2 = guess2 + inpow;
                tab = 2;
                }
              range = pow10((double)guess2)*range;
               fill = random(1000)*0.001*range;
               fill = floor(fill);
              if(tab ==1)
                     fill2 = fill/pow10(3)+min[index2];
              else if(tab ==2)
                  fill2 = (int)fill/pow10(inpow) + min[index2];
          p[index][index2] = fill2;

        }
    }
 }

 void filhand(void)
 {
 FILE *fptr;
 char name[10];
 int x;
 int tempcount;
  clrscr();
   numdata =0;

  if((fptr=fopen("k5.txt","r"))== NULL)
       {
         printf("sorry cannot open this file\n");
         exit(1);
        }


while(fscanf(fptr,"%f%f%f%f%f%f%f%f",&enh[numdata],&ena[numdata],&ek[numdata
],&eca[numdata],

&anh[numdata],&ana[numdata],&ak[numdata],&aca[numdata]) != EOF)
        {
        /*
printf("%f,%f,%f,%f\n",enh[numdata],ena[numdata],ek[numdata],eca[numdata]);

printf("%f,%f,%f,%f\n",anh[numdata],ana[numdata],ak[numdata],aca[numdata]); */
       numdata = numdata +1;
      }
  fclose(fptr);
  printf("NUMBER OF DATAPOINTS %d\n",numdata);

}

/**** end of file handling function ****/
```

```
/*** expansion contraction sequence ***/

double amotry(float fac)
    {

       int j;

       float fac1,fac2;
    double ytry;
       fac1 = (1.0-fac)/ndim;
       fac2 = fac1-fac;

       for(j=1;j<=ndim;j++)
           {
           ptry[j] = psum[j]*fac1-p[ihi][j]*fac2;
           if (ptry[j] <0)
                  ptry[j] = fabs(ptry[j]);
           }
       ytry = calcfunc(ptry);
       if(ytry <y[ihi])
         {
         y[ihi] = ytry;
         for(j=1;j<=ndim;j++)
           {
           psum[j] = psum[j] + ptry[j]-p[ihi][j];
           p[ihi][j] = ptry[j];
           }

         }
       return(ytry);
    }


    /*************************************************************/
    /*  EVALUATING THE FITNESS OF THE VERTICES OF THE SIMPLEX   */
    /*************************************************************/

    void evalvertices(void)
        {
        int c,tempcount,tempcount2;
         averagefitness = 0.0;

      for(tempcount=1;tempcount<=mpts;tempcount++)
          for(tempcount2 =1;tempcount2<=ndim;tempcount2++)
            {
               /* if(p[tempcount][tempcount2] < 0)
                  {
                   printf("vertex %d dimension %d value %ld\n",tempcount,
tempcount2,p[tempcount][tempcount2]);
```

```
            exit(0);
                } */
        }

    for(c=1;c<=mpts;c++)
      {

       fitness[c] = calcfunc(*(p+c));
       averagefitness = averagefitness + fitness[c];
      }
      averagefitness = averagefitness/mpts;

    }
  /****** END OF EVALUATION FUNCTION
****************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
FILE *fptr1;
FILE *fptr2;
FILE *fptr3;

double actnh[100],actna[100],actk[100],actca[100];
double a[50][50],b[50][50];
double enh[100],ena[100],ek[100],eca[100];
double emodnh[100],emodna[100],emodk[100],emodca[100];
double actnew[100];
double parm[10][10];

int numact,numpot;

void filing(void);
void gauss(void);
void correlation(void);
void fil2(void);
void main(void)
 {
 int count;
 double temp;

 filing();

 /*** setting up initial N-K parameters ***/
 parm[1][1] = 225.775836;
 parm[1][2] = 45.443407;
 parm[1][3] = 1.0;
 parm[1][4] = 9.36e-4;
 parm[1][5] = 0.184931;
 parm[1][6] = 0.006465;

 parm[2][1] = 257.189705;
 parm[2][2] = 54.842883;
 parm[2][3] = 1.0e-7;
 parm[2][4] = 1.0;
 parm[2][5] = 0.106890;
 parm[2][6] = 0.017226;

 parm[3][1] = 270.155439;
 parm[3][2] = 51.395680;
 parm[3][3] = 0.143042;
 parm[3][4] = 1.0e-7;
 parm[3][5] = 1.0;
 parm[3][6] = 0.005359;

 parm[4][1] = 119.0813;
```

```
 parm[4][2] = 23.293456;
 parm[4][3] = 1e-7;
 parm[4][4] = 0.000897;
 parm[4][5] = 0.001672;
 parm[4][6] = 1.0;


/*****************************************************/
/* Converting from bare potentials to alpha terms where alpha      */
/* = (Emeas - Ecell)/slope                                          */
/*****************************************************/


 for(count=1;count<=numpot;count++)
   {
     b[1][count] = pow(10.0,(enh[count]-parm[1][1])/parm[1][2]);
     b[2][count] = pow(10.0,(ena[count]-parm[2][1])/parm[2][2]);
     b[3][count]  = pow(10.0,(ek[count] -parm[3][1])/parm[3][2]);
     b[4][count] = pow(10.0,(eca[count]-parm[4][1])/parm[4][2]);
   }

gauss();
 printf("help \n");
/* getch();  */
 correlation();
 fil2();
/* getch(); */
 }

void filing(void)
 {
  numpot =1;
  if((fptr1=fopen("070593a.xls","r"))== NULL)
   {
    printf("sorry cannot open the activity file \n");
    exit(1);
    }
  if((fptr2=fopen("270593a.xls","r"))== NULL)
   {
    printf("sorry cannot open the potential file");
    exit(1);
    }


while((fscanf(fptr1,"%lf%lf%lf%lf",&actnh[numpot],&actna[numpot],&actk[numpot],&
actca[numpot])) != EOF)
      {
          numpot++;
          printf("%d\n",numpot);
       }
```

```
    printf("entered activities \n");
    numact=1;

while((fscanf(fptr2,"%lf%lf%lf%lf",&enh[numact],&ena[numact],&ek[numact],&eca[nu
mact])) != EOF)
     {
        numact++;
        printf("%d\n",numact);
      }

  numpot = numpot-1;

 fclose(fptr1);
 fclose(fptr2);
 }


void gauss(void)
 {

  int i,icol,irow,j,k,l,ll,n,m;
  double big,dum;
  double pivinv;
  int indxc[50],indxr[50],ipiv[50];

  n=4;
  m =numpot;
  for(i=1;i<=n;i++)
   {
    for(j=1;j<=4;j++)
        a[i][j] = parm[i][2+j];
   }
 /*  n=4;
   m =1;
   a[1][1] = 1.0;
   a[1][2] = 1.0;
   a[1][3] = 1.0;
   a[1][4] = 1.0;

   a[2][1] = 1.0;
   a[2][2] = 1.0;
   a[2][3] = 2.0;
   a[2][4] = 1.0;

   a[3][1] = 2.0;
   a[3][2] = 1.0;
   a[3][3] = 1.0;
   a[3][4] = 1.0;

   a[4][1] = 1.0;
```

```
   a[4][2] = 2.0;
   a[4][3] = 1.0;
   a[4][4] = 1.0;   */
    for(j=1;j<=4;j++)
   ipiv[j] =0;


for(i=1;i<=n;i++)
 {
    big=0;
    for(j=1;j<=n;j++) /* move thru all the rows to find the pivot element */
        {
        if(ipiv[j] !=1)
          {
          for(k=1;k<=n;k++)  /****-----------*/
            {                        /*** move thru' all the   ***/
             if(ipiv[k] ==0)              /*** elements of a given  ***/
                {                        /*** row to find the      ***/
                 if(fabs(a[j][k])>= big)      /*** biggest element of   ***/
                   {                     /*** row.                 ***/
                   big = fabs(a[j][k]);           /**/
                   irow = j;                     /**/
                   icol = k;                     /**/
                   }                            /**/
                 }                              /**/
             else if(ipiv[k] >1)                 /**/
                 {                              /**/
                   printf("error singular matrix \n"); /**/
                   exit(1);                     /**/
                 }                              /**/
            }  /****--------------------------------*/

          } /*** end of if ipiv[j] ***/
        } /*** end of j for-next loop ***/

        ipiv[icol] = ipiv[icol] +1;

        /*** got the pivot element now to swap rows to put it on ***/
        /*** the diagonal need to keep check of indices for further **/
        /*** unscrambling at the end ****/


        if(irow != icol)
          {
          for(l=1;l<=n;l++)
                {
                 dum = a[irow][l];
                 a[irow][l] =a[icol][l];
                 a[icol][l] = dum;
```

```
            }
          for(l=1;l<=m;l++)
            {
                dum = b[irow][l];
                b[irow][l] = b[icol][l];
                b[icol][l] = dum;
          }
          }
        indxr[i] = irow;
        indxc[i] = icol;

        if(a[icol][icol] == 0.0)
          {
           printf("singular matrix.... zero on the diagonal \n");
           exit(1);
          }

        pivinv = 1.0/(a[icol][icol]);  /** setting up division factor for the reduction of
the other lines  ***/

        a[icol][icol] =1.0;

        for(l=1;l<=n;l++)
          a[icol][l] = a[icol][l]*pivinv;

        for(l=1;l<=m;l++)
          b[icol][l] = b[icol][l]*pivinv;

        for(ll=1;ll<=n;ll++)
          {
          if(ll != icol)
            {
                dum = a[ll][icol];
                a[ll][icol] = 0.0;
                for(l=1;l<=n;l++)
                   a[ll][l] = a[ll][l] - a[icol][l]*dum;
                for(l=1;l<=m;l++)
                   b[ll][l] = b[ll][l] -b[icol][l]*dum;
            }
          }
     }  /*** end of reduction section ***/

    /*** now for unscrambling  !! **/

    for(l=n;l>=1;l--)
      {
         if(indxr[l] !=indxc[l])
            {
             for(k=1;k<=n;k++)
                 {
```

```
                        dum = a[k][indxr[l]];
                        a[k][indxr[l]] = a[k][indxc[l]];
                        a[k][indxc[l]] = dum;
                       }
                  }
             }


}

void correlation(void)
 {
  double r;
  double meannhx,meannhy,sumx,sumy;
  double meannax,meannay,meankx,meanky;
  double meancax,meancay;
  double snhx,snax,skx,scax;
  double snhy,snay,sky,scay;

  double sqnhx,sqnax,sqkx,sqcax;
  double sqnhy,sqnay,sqky,sqcay;
  double sqnhxy,sqnaxy,sqkxy,sqcaxy;

  double rnh,rna,rk,rca;
  int m,n;

  for(m=1;m<=numpot;m++)
   {
    for(n=1;n<=4;n++)
        {
        if(n==1)
          emodnh[m] = b[n][m];
        if(n==2)
           emodna[m] = b[n][m];
        if(n==3)
           emodk[m] = b[n][m];
        if(n==4)
           emodca[m] = b[n][m];
        }
    }

   meannhx=0.0;
   meannhy=0.0;
   meannax=0.0;
   meannay=0.0;
   meankx=0.0;
   meanky=0.0;
   meancax=0.0;
   meancay=0.0;
```

```
for(m=1;m<=numpot;m++)
  {
     meannhx= actnh[m]+meannhx;
     meannhy = meannhy + emodnh[m];
     meannax = meannax + actna[m];
     meannay = meannay + emodna[m];
     meankx = meankx + actk[m];
     meanky = meanky + emodk[m];
     meancax = meancax + actca[m];
     meancay = meancay + emodca[m];
     }
     meannhx=meannhx/numpot;
     meannhy =meannhy/numpot;
     meannax = meannax/numpot;
     meannay = meannay/numpot;
     meankx = meankx/numpot;
     meanky = meanky/numpot;
     meancax = meancax/numpot;
     meancay = meancay/numpot;

sqnhx =0.0;
sqnhy =0.0;
sqnax =0.0;
sqnay =0.0;
sqkx = 0.0;
sqky = 0.0;
sqcax = 0.0;
sqcay = 0.0;
sqnhxy = 0.0;
sqnaxy = 0.0;
sqkxy =0.0;
sqcaxy = 0.0;

for(m=1;m<=numpot;m++)
   {
    snhx = actnh[m] - meannhx;
    snhy = emodnh[m] -meannhy;
    snax = actna[m] -meannax;
    snay = emodna[m] -meannay;
    skx = actk[m] - meankx;
    sky = emodk[m] -meanky;
    scax = actca[m] - meancax;
    scay =emodca[m] -meancay;

    sqnhx = sqnhx +(snhx*snhx);
    sqnhy = sqnhy +(snhy*snhy);
    sqnhxy = sqnhxy +(snhx*snhy);

    sqnax = sqnax +(snax*snax);
    sqnay = sqnay +(snay*snay);
```

```
      sqnaxy = sqnaxy + (snax*snay);

      sqkx = sqkx +(skx*skx);
      sqky = sqky +(sky*sky);
      sqkxy = sqkxy + (sky*skx);

      sqcax = sqcax +(scax*scax);
      sqcay = sqcay +(scay*scay);
      sqcaxy = sqcaxy +(scax*scay);
     }

    rnh =sqnhxy/sqrt(sqnhx*sqnhy);
    rna =sqnaxy/sqrt(sqnax*sqnay);
    rk = sqkxy/sqrt(sqkx*sqky);
    rca = sqcaxy/sqrt(sqcax*sqcay);


    printf("Ammonium ISE conc. correlation %lf\n",rnh);
    printf("Sodium ISE conc. correlation %lf\n",rna);
    printf("Potassium ISE conc. correlation %lf\n",rk);
    printf("Calcium ISE conc. correlation %lf\n",rca);

  }

void fil2(void)
  {
  FILE *fptr3;
  int count;

  if((fptr3=fopen("GA1953.txt","w"))==NULL)
    {
     printf("sorry cannot open exit file \n");
     exit(1);
    }
  for(count=1;count<=numpot;count++)
   {
    fprintf(fptr3,"%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf \n",actnh[count],emodnh[count],
            actna[count],emodna[count],actk[count],emodk[count],
            actca[count],emodca[count]);
    }
    fclose(fptr3);
    }
```

Gastab.c

```
#include <stdlib.h>
#include <stdio.h>
#include <alloc.h>
#include <math.h>
#include <string.h>

#include "randhead.c"
/*#include "equil4.c"   */
#define RIJ 4
#define KOLOM 4
#define NMAX 20
#define MMAX 20
#define LMAX  10
#define BETA 0.5
#define GAMMA 2.0
#define ALPHA 1.0
#define R 8.31441
#define F 96485.309
#define BUFSIZE 50001
#define NUMCHROM 300
#define MAXPARM 20
#define MAXSPECIES 10
#define MAXEQUIL 10
#define MAXLIG 5
#define MAXMETAL 4
#define WORDSIZE 16
#define MAXWORD 65535
#define MAXDATA 50
#define CROSSPOOL 130
#define CROSSRATE 0.65
#define NUMELEC 2
#define T 298.15
#define SCONV R*T/(NUMELEC*F)
#define CHARGEMETAL 2
#define CHARGELIGAND 0

struct chromosomedata
 {
  int prop[2];
  int modord[MAXMETAL+MAXLIG];
  int parm;
  unsigned int *gene;
  double fitness;
 };

struct chromosomedata *chromosome[NUMCHROM];
int numdata=8;
double maxstab,minstab,averagefitness;
double test,maxheat,minheat;
double *totmet,*ligconc,*Qcp,*mltit;
```

```
int *choice,*geneindex;
char symb[LMAX] ;
int prec ;
int coeff;
int nchar;
char tsymb[MMAX][LMAX];
double v[NMAX][MMAX] ;
int tprec[MMAX], ordec0[MMAX];
int nequil,mequil, m0;
int signum ;

double davis(double ionicstr,int charge);
void fillpool(void);
void select(void);
void evalgenes(void);
void calcfunc(int chromind);
void filhand(void);
double getchromosome(unsigned int s);
double concdet(double **beta, double MT, double LT,int chromind,double *p);
double *equilcalc(double **beta,int chromind,int dataind);
void qsortc(int l, int r,double *bestfitness);
void sort(void);
void rankscale(void);
void preselect(void);
void mutategene(void);
void cross(int cpoint,unsigned int gene1,unsigned int gene2,unsigned int
*temp1,unsigned int *temp2);
void reject(void);
int hamming(int chrom1,int chrom2);
double getchromosomeheat(unsigned int s);
void detsym(double *,int nequil,double *);
void solsym(double *, int, double *);
void guess(int, int, double *, double *, double *, double *);
void iter(int , int, double *, double *, double *, double *);
double *diag( double *, int, int);
double *elem( double *, int, int, int);
double mina(double, double);
double maxa(double, double);
void fault(int);
void readc(void);
void probeer(double *, int, int);
void name(void);
void term(void);
void side(void);
char *index(register const char *,register const char);

void main(void)
{
 int numiter,i,j,flag,report,try,filing,midway;
 int step=5;
```

```
 FILE *fptr2;
 float proby;
 struct chromosomedata *tempptr;
 /* printf("initial amout of memory is %lu/n",(unsigned long)coreleft());*/
 for(i=0;i<NUMCHROM;i++)
  {
   if((chromosome[i]=(struct chromosomedata *)malloc(sizeof(struct
chromosomedata)))==NULL)
    {
     printf("problems establishing the population  structures \n");
     exit(1);
     }
  }

if((ligconc=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
    printf("problems allocating the ligand concentration \n");
    exit(1);
  }
 printf("delatE allocation \n");

 if((totmet=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
   printf("problems establishing the total metal matrix\n");
   exit(1);
  }
printf("Qcp allocation \n");

if((Qcp=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
 {
  printf("problems allocating the peak current \n");
  exit(1);
 }
printf("mltit allocation \n");

if((mltit=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
 {
  printf("problems allocating the peak current \n");
  exit(1);
 }

printf("geneindex allocation \n");
if((geneindex=(int *)malloc(NUMCHROM*sizeof(int)))==NULL)
 {
  printf("problems establishing the geneindex \n");
  exit(1);
  }

 printf("choice allocation \n");
if((choice=(int *)malloc(NUMCHROM*sizeof(int)))==NULL)
```

```
{
 printf("problems establishing the selection index \n");
 exit(1);
}

for(i=0;i<NUMCHROM;i++)
 for(j=0;j<2;j++)
  (chromosome[i])->prop[j]=1;

if((fptr2=fopen("calthio.csv","a"))==NULL)
  {
  printf("cannot open the output data file \n");
  exit(1);
  }
 /* printf("initial allocation memory left %lu/n",(unsigned long)coreleft());    */

 printf("setting up the random function \n");
 setrand();
 filhand();
 fillpool();
/* printf("filled pool coreleft %lu/n",(unsigned long)coreleft());*/

 evalgenes();
sort();
for(numiter=0;numiter<300;numiter++)
  {
   printf("iteration %d\n",numiter);
   rankscale();
    preselect();
    select();
    sort();
    proby=randnew();
   if(proby<CROSSRATE)
    {
               reject();
}

   mutategene();
evalgenes();

sort();

   printf("averagefitness %lf\n",averagefitness);
   printf("most fit chromosome\n");
   try=*(geneindex+0);
   midway=0.5*(chromosome[try])->parm;
   printf("fitness %lf\n",(chromosome[try])->fitness);
   printf("model order metal %d ligand %d/n",(chromosome[try])-
>modord[0],(chromosome[try])->modord[1]);
   if((numiter%step)==0.0)
```

```
        fprintf(fptr2,"%d,%lf,%lf,",numiter,(chromosome[try])->fitness,averagefitness);

     for(report=0;report<midway;report++)
      {
        printf("g %d v %lf",report,getchromosome(*((chromosome[try])-
>gene+report)));
        if((numiter%step)==0.0)
          fprintf(fptr2,"%lf,",getchromosome(*((chromosome[try])->gene+report)));
      }

     for(report=midway;report<(chromosome[try])->parm;report++)
      {
        printf("g %d v %lf",report,getchromosomeheat(*((chromosome[try])-
>gene+report)));
        if((numiter%step)==0.0)
          fprintf(fptr2,"%lf,",getchromosomeheat(*((chromosome[try])->gene+report)));
      }

    fflush(fptr2);
    if((numiter%step)==0.0)
     {
      fprintf(fptr2,"\n");
      fprintf(fptr2,"others\n");
      for(filing=1;filing<NUMCHROM;filing++)
          {
           try=*(geneindex+filing);
           midway=0.5*(chromosome[try])->parm;
           fprintf(fptr2,"%lf,%d,%d,",(chromosome[try])->fitness,(chromosome[try])-
>modord[0],(chromosome[try])->modord[1]);
           for(report=0;report<midway;report++)
             fprintf(fptr2,"%lf,",getchromosome(*((chromosome[try])->gene+report)));
           for(report=midway;report<(chromosome[try])->parm;report++)
              fprintf(fptr2,"%lf,",getchromosomeheat(*((chromosome[try])-
>gene+report)));
           fflush(fptr2);
           fprintf(fptr2,"\n");
          }
      }/*end of if..*/
   printf("\n\n");
  }
   fclose(fptr2);
}
/*************************************************************************
**/
/************** FILE HANDLING FUNCTION
******************************/
void filhand(void)
{
 FILE *fptr;
 int tempcount;
```

```
 char comma;
/* double templig[MAXDATA],tempE[MAXDATA],tempi[MAXDATA]; */
/* clrscr();*/
 numdata =0;

 if((fptr=fopen("newcrow3.csv","r"))== NULL)
         {
          printf("sorry cannot open the data file\n");
          exit(1);
         }
while(fscanf(fptr,"%lf%c%lf%c%lf%c%lf",mltit+numdata,&comma,ligconc+numdata,&
comma,totmet+numdata,&comma,Qcp+numdata) != EOF)
  {
   printf("data
%lf,%lf,%lf,%lf\n",*(mltit+numdata),*(ligconc+numdata),*(totmet+numdata),*(Qcp+nu
mdata));
   numdata = numdata +1;
  }

 fclose(fptr);

}
/************* end of file handling function ********************************/


/****************** FILLING THE GENEPOOL
****************************/
void fillpool(void)
{
 int i,index,index2;
 int metallim[MAXMETAL],liglim[MAXLIG];
 double roundtempi,roundtempf ;
 double tempcounter;
 unsigned int tempgene;
 int metal,ligand,genesize;

 /* metallim[0]=1;
  liglim[0]=3; */
  maxstab=15.0;
  minstab=0.0;
  maxheat=5.0;
  minheat=-5.0;

 for(index=0;index<NUMCHROM;index++)
  {
    metal=2;
    (chromosome[index])->modord[0]=metal;
    ligand=1;
    (chromosome[index])->modord[1]=ligand;
```

```
    /* considers length of gene sequence with respect to */
    /* stability constants  and heats of formation */
    genesize=ligand*metal; /* +(metal-1) +(ligand-1); self-complexation */
    genesize=genesize*2.0; /* to consider heats of formation */


    (chromosome[index])->parm=genesize;

    (chromosome[index])->gene=(unsigned int *)malloc(genesize*sizeof(unsigned int));
    if((chromosome[index])->gene==NULL)
     {
      printf("not enough memory to initialise the genepool \n");
      exit(1);
     }

    /*** initialising genepool with gene sequences corresponding */
    /*** to stability constants                                   */
    for(index2=0;index2<(chromosome[index])->parm;index2++)
      {
         tempcounter=(double)randnew()*MAXWORD;
         roundtempf= modf(tempcounter,&roundtempi);
         if(roundtempf >0.5)
            roundtempi++;
         tempgene=(unsigned int)roundtempi;
         *((chromosome[index])->gene+index2)=tempgene;
      }
    (chromosome[index])->fitness=0.0;
}

}
/**************** end of filling the population ***************************/

/* PRESELECTING CHROMOSOMES FOR BREEDING BY STOCHASTIC
REMAINDER SAMPLING */

void preselect(void)
{
 int j,k,jassign,winner;
 static int alreadydone=0;
 static float *fraction;
 double expected;


 if(!alreadydone)
  {
   if((fraction=(float *)malloc(NUMCHROM*sizeof(float)))==NULL)
    {
     printf("problems establishing the fractional geneindex \n");
     exit(1);
    }
```

```
   alreadydone=1;
  }

 j=0;
 k=0;
 for(j=0;j<NUMCHROM;j++)
  {
   expected = (chromosome[*(geneindex+j)])->fitness;
   jassign = (int)floor(expected);
   *(fraction+j) = expected - jassign;

   while(jassign >0)
    {
    jassign = jassign -1;
     *(choice+k) = j;
    k++;
    }
  }

 j=0;

 while(k < NUMCHROM)
  {
  if(j>=NUMCHROM)
   j=0;   /* resetting to the start of the population if it overruns without filling the
temporary pool */

  if(*(fraction+j) >0.0)
   {
    if(flip(*(fraction+j)) ==1)
         winner =1;
    else winner =0;
   if(winner ==1)
    {
     *(choice+k) =j;
     *(fraction+j) = *(fraction+j) -1.0;
     k++;
    }
   }
  j++;
  }

}

/**** end of stochastic remainder presampling function ******************/


/************ SELECTION OF CHROMOSOMES FOR BREEDING
*********************/
```

```
void select()
 {

  static struct chromosomedata *tempool[NUMCHROM];
  static int alreadydone =0;
  int i,j,temp,genespace,genespace2;
  int k,jpick;
  int nremain;
  int l,hold;

  if(!alreadydone)
   {
    for(i=0;i<NUMCHROM;i++)
     {
      if((tempool[i]=(struct chromosomedata *)malloc(sizeof(struct
chromosomedata)))==NULL)
       {
         printf("problems establishing the temp breeding population  structures \n");
         exit(1);
       }
     }
    alreadydone=1;
   }

  nremain=NUMCHROM;

  for(i=0;i<NUMCHROM;i++)
   {
    jpick=pickrand(0,nremain-1);
    temp= *(geneindex+ *(choice+jpick));
    genespace=(chromosome[temp])->parm;
    (tempool[i])->gene=(unsigned int *)malloc(genespace*sizeof(unsigned int));
    if((tempool[i])->gene==NULL)
      {
       printf("not enough memory to allocate to the breeding population \n");
       exit(1);
       }
     (tempool[i])->parm=genespace;
     (tempool[i])->fitness=(chromosome[temp])->fitness;

     for(k=0;k<genespace;k++)
       *((tempool[i])->gene+k)=*((chromosome[temp])->gene+k);

     for(k=0;k<2;k++)
      {
       (tempool[i])->modord[k]=(chromosome[temp])->modord[k];
       (tempool[i])->prop[k]=(chromosome[temp])->prop[k];
       }

     hold = *(choice+jpick);
```

```
    *(choice+jpick) = *(choice+nremain-1);
    *(choice+nremain-1) = hold;
    nremain = nremain -1;
   }


  for(i=0;i<NUMCHROM;i++)
   {
    genespace2=(tempool[i])->parm;
    if((chromosome[i])->parm <genespace2)
     {
        (chromosome[i])->gene=(unsigned int *)realloc((void *)(chromosome[i])-
>gene,genespace2*sizeof(unsigned int));
        if((chromosome[i])->gene==NULL)
         {
         printf("cannot reallocate \n");
         exit(1);
         }
       }
    (chromosome[i])->parm=genespace2;
    (chromosome[i])->fitness=(tempool[i])->fitness;

    for(k=0;k<genespace2;k++)
        *((chromosome[i])->gene+k)=*((tempool[i])->gene+k);

    for(k=0;k<2;k++)
     {
       (chromosome[i])->modord[k]=(tempool[i])->modord[k];
       (chromosome[i])->prop[k]=(tempool[i])->prop[k];
     }

   }

  for(i=0;i<NUMCHROM;i++)
   free((tempool[i])->gene);

}

/***************************************************************************
*/
void evalgenes()
{
 int c,i,tempcount,tempcount2;

 averagefitness=0.0;

 for(c=0;c<NUMCHROM;c++)
  {
   /* printf("chromosome %d\n",c);    */
       calcfunc(c);
```

```
    averagefitness = averagefitness + (chromosome[c])->fitness;
  /* printf("fitness %lf\n",(chromosome[c])->fitness); */

        }
 averagefitness = averagefitness/NUMCHROM;
}
/****** end of chromosome evaluation function ****************************/

/**** DETERMINATION OF MSE FOR A GIVEN CHROMOSOME  ***********/
void calcfunc(int chromind)
{
 double *beta[MAXMETAL];
 double *concarr;
 double actcoef[MAXSPECIES];
 int betasize,i,betaind,datacounter,tempactpos;
 int j,metalsize,ligsize,tailend,geneptr,enthalpyindex;
 int metind,ligind,numspecies;
 int charge[MAXSPECIES];
 double metalactcoef,ligactcoef;
 double error,metal,sum,calce,expte,metalconc,thermconv;
 double ionicstrength,ionicstr2,errionic,newtitvolume,vesselvolume;
 unsigned int temphold;
 double base=10.0;

 /*printf("before allocation memory %lu\n",(unsigned long)coreleft()); */

 for(i=0;i<=(chromosome[chromind])->modord[0];i++)   /*self-complexation */
  {
   betasize=(chromosome[chromind])->modord[1] +1;
   if((beta[i]=(double *)malloc(betasize*sizeof(double)))==NULL)
    {
     printf("problems establishing the beta array \n");
     exit(1);
    }
  }

/*  printf("after allocation %lu",(unsigned long)coreleft());   */

for(i=0;i<=(chromosome[chromind])->modord[0];i++)
   for(j=0;j<(chromosome[chromind])->modord[1]+1;j++)
        *(beta[i]+j)=1.0;

  *(beta[0]+0)=0.0;  /*nothing present for complexation */

  metalsize=(chromosome[chromind])->modord[0];
  ligsize=(chromosome[chromind])->modord[1];
  tailend=metalsize*ligsize;
  metind=1;
  ligind=1;
```

```
 for(i=0;i<tailend;i++)
  {
   if(ligind>(chromosome[chromind])->modord[1])
    {
     ligind=1;
     metind++;
    }
   temphold=*((chromosome[chromind])->gene+i);
   *(beta[metind]+ligind)=getchromosome(temphold);
   ligind++;
 }

numspecies=(chromosome[chromind])->modord[0] * (chromosome[chromind])-
>modord[1];
  numspecies= numspecies + (chromosome[chromind])->prop[0] +
(chromosome[chromind])->prop[1];
  vesselvolume=2.5087;
  error=0.0;

  if((CHARGEMETAL !=0) || (CHARGELIGAND !=0))
   { charge[0]=CHARGEMETAL;
    charge[1]=CHARGELIGAND;
    metind=1;
    ligind=1;
    for(j=2;j<numspecies;j++)
     {
      if(ligind >(chromosome[chromind])->modord[1])
          {
          metind++;
          ligind =1;
          }
    charge[j]=metind*CHARGEMETAL +ligind*CHARGELIGAND;
    ligind++;
     }
    }

  for(datacounter=1;datacounter<numdata;datacounter++)
   {
    metalconc=*(totmet+datacounter);
    concarr=equilcalc(beta,chromind,datacounter);
    if((CHARGEMETAL !=0) || (CHARGELIGAND !=0))
     {
        ionicstrength=0.0;
        for(i=0;i<numspecies;i++)
          ionicstrength= ionicstrength+ *(concarr+i)*charge[i]*charge[i];
        ionicstrength=ionicstrength + 2.0*metalconc; /* to take into account acetate */
        ionicstrength=0.5*ionicstrength;

     /* Loop for adjusting the thermodynamic equilibrium constant to a */
     /* concentration equilibrium constant by iteratively reducing the */
```

```
/* difference between ionic strength values from each iteraion    */

do
   {
    for(i=0;i<numspecies;i++)
     {
      if(charge[i] != 0.0)
       {
          actcoef[i]=1.0+sqrt(ionicstrength);
          actcoef[i]=sqrt(ionicstrength)/actcoef[i];
          actcoef[i]=actcoef[i] -0.2*ionicstrength;
          actcoef[i]=actcoef[i]*((double)charge[i]);
          actcoef[i]=actcoef[i]*((double)charge[i]);
          actcoef[i]= actcoef[i] * (-0.512);
          actcoef[i]=pow(base,actcoef[i]); ;
         }
      else
        actcoef[i]=1.0;
     }
   tempactpos =0;
   for(i=1;i<=(chromosome[chromind])->modord[0];i++)
     for(j=1;j<=(chromosome[chromind])->modord[1];j++)
      {
       metalactcoef=actcoef[0];
       ligactcoef=actcoef[1];
       thermconv= actcoef[tempactpos+2];
       thermconv=thermconv/pow(metalactcoef,(double)i);
       thermconv=thermconv/pow(ligactcoef,(double)j);
       thermconv=log10(thermconv);
       *(beta[i]+j)=*(beta[i]+j)+thermconv;
       tempactpos++;
      }

concarr=equilcalc(beta,chromind,datacounter);
ionicstr2=0.0;
for(i=0;i<numspecies;i++)
    ionicstr2= ionicstr2 + *(concarr+i)*charge[i]*charge[i];
ionicstr2=ionicstr2 + 2.0*metalconc; /* to take into account acetate */
ionicstr2=0.5*ionicstr2;

errionic=(ionicstrength-ionicstr2)*(ionicstrength-ionicstr2);

ionicstrength=ionicstr2;

/* resetting the stability constant matrix */

for(i=0;i<=(chromosome[chromind])->modord[0];i++)
   for(j=0;j<(chromosome[chromind])->modord[1]+1;j++)
      *(beta[i]+j)=1.0;
```

```
    *(beta[0]+0)=0.0;   /*nothing present for complexation */

    metalsize=(chromosome[chromind])->modord[0];
    ligsize=(chromosome[chromind])->modord[1];
    tailend=metalsize*ligsize;
    metind=1;
    ligind=1;

    for(i=0;i<tailend;i++)
     {
       if(ligind>(chromosome[chromind])->modord[1])
        {
         ligind=1;
         metind++;
        }
     temphold=*((chromosome[chromind])->gene+i);
     *(beta[metind]+ligind)=getchromosome(temphold);
     ligind++;
    }

}while(errionic >1e-10);
   }
  /* conversion of conc. M to moles corrected for changing volume of */
  /* soln. in the titration vessel                               */

  newtitvolume=vesselvolume+ *(mltit+datacounter);
  for(i=0;i<numspecies;i++)
    *(concarr+i)=*(concarr+i)*newtitvolume* 0.001;

  /* summation of enthalpies of formation/moles product for each species */
  /* formed by the reactions studied. Since the metal and ligand are not */
  /* formed by the forward equilibrium reaction their contribution to    */
  /* the sum will not be considered                                   */
  /* The genes encoding the enthalpy of formation follow those encoding  */
  /* the stability constants, so need to shift the gene index to start   */
  /* position of enthalpy constants on the chromosome                   */

  sum=0.0;

  for(i=2;i<numspecies;i++)
   {
       enthalpyindex=(chromosome[chromind])->parm/2.0;
       enthalpyindex=enthalpyindex +(i-2);
       temphold=*((chromosome[chromind])->gene+enthalpyindex);
       sum=sum+ *(concarr+i)*getchromosomeheat(temphold)*100000.0;
   }
   error=error +(*(Qcp+datacounter)-sum)*(*(Qcp+datacounter)-sum);
} /* end of datacounter loop */

(chromosome[chromind])->fitness=error;
```

```
for(i=0;i<=(chromosome[chromind])->modord[0];i++)
     free(beta[i]);
}
/*********************************************************************
*/

/********** GETTING THE UNSCALED VALUE OF A GENE (stability constant)
**/
double getchromosome(unsigned int s)
 {
  double scalefact;
  scalefact=1.0/MAXWORD;
  scalefact = (double)s*scalefact;
  scalefact = scalefact* (maxstab - minstab) + minstab;
  return(scalefact);
 }
/*************** end of conversion function ********************************/

/********** GETTING THE UNSCALED VALUE OF A GENE (heat of formation)
**/
double getchromosomeheat(unsigned int s)
 {
  double scalefact;
  scalefact=1.0/MAXWORD;
  scalefact = (double)s*scalefact;
  scalefact = scalefact* (maxheat - minheat) + minheat;
  return(scalefact);
 }
/*************** end of conversion function ********************************/

/**************** QUICKSORTING PROCEDURE
***************************/
void sort(void)
{
 int i,j,k;
 static int alreadydone=0;
 static double *bestfitness;


 if(!alreadydone)
   {
         if((bestfitness=(double *)malloc(NUMCHROM*sizeof(double)))==NULL)
          {
         printf("problems establishing the bestfitness array for sorting \n");
         exit(1);
          }
         alreadydone=1;
   }
```

```
 for(j=0;j<NUMCHROM;j++)
  {
        *(geneindex+j) = j;
        *(bestfitness+j)= (chromosome[j])->fitness;
        }

 qsortc(0,NUMCHROM-1,bestfitness);
 }

void qsortc(int l, int r,double *bestfitness)
{
 int i,j;
 double x,y;
 int d_index,pos;
 div_t cent;

 i=l;
 j=r;
 cent = div((l+r),2);
 pos = cent.quot;
 x =*(bestfitness+pos);                  /* FITNESS OF MIDDLE OF THE POPN.*/

 do
  {
        while(*(bestfitness+i) < x && i< r) i++;
          while(x < *(bestfitness+j) && j> l) j--;
        if(i<=j)
          {
                y= *(bestfitness+i);
        *(bestfitness+i) = *(bestfitness+j); /*the values of the objective function are
sorted */
                *(bestfitness+j) = y;
                d_index= *(geneindex+i); /* but the genes themselves are not sorted
rather */
                *(geneindex+i) = *(geneindex+j);   /*** indices are sorted ***/
                *(geneindex+j) = d_index;
                i= i+1;
                j= j-1;
          }

 }
 while(i<=j);

 if(l<j) qsortc(l,j,bestfitness);
 if(i<r) qsortc(i,r,bestfitness);

 }

/********************** end of quicksorting routine **********************/
```

```
/** SCALING CHROMOSOMES ACCORDING TO THEIR RANK AND
INTEGRAL */
void rankscale(void)
{
 int count,i;
 double m,c;
 double uplim,sumfitness;

 uplim = 1.1;
 m=2.0*(1.0-uplim)/NUMCHROM;
 c= uplim;
 sumfitness = 0.0;
 for(count=0;count<NUMCHROM;count++)
   {
        (chromosome[*(geneindex+count)])->fitness= count*m + c;
        sumfitness = sumfitness + (chromosome[*(geneindex+count)])->fitness;
   }
}
/**************** end of rank scaling procedure ************************/


/*********************************************************************
*/
/*          GENE MUTATION OPERATOR FUNCTION                    */
/*********************************************************************
/

void mutategene(void)
{
 /* Uses XOR to flip different bits. Mutation rate is 1/population size. */
 /* Indices are used to describe which bit position mutation is to occur */
 /* and also the chromosomes on which it will occur                    */

 int mutepoint,atchrom,k,tempcount,tempcount2;
 int genesize,prevent;
 unsigned int tempmeas;
 double c,num =2.0;
 double roundfmut,roundfat,roundimut,roundiat;
 float proby,muterate;

 muterate =1.0/NUMCHROM;

 for(k=0;k<NUMCHROM-1;k++)
   {
        prevent=*(geneindex+k);
        proby = randnew();
        if(proby < muterate)
          {
           proby = randnew();
           roundfmut= modf((double)(proby*(WORDSIZE-1)),&roundimut);
```

```
        if(roundfmut>0.5)
                roundimut=roundimut+1;          /* BIT POSITION */

        proby = randnew();

        genesize=(chromosome[prevent])->parm;
        roundfat= modf((double)(proby*(genesize-1)),&roundiat);
        if(roundfat>0.5)
       roundiat=roundiat++;

        /******* PERFORM XOR OPERATION *****/
        c= pow(num,(double)roundimut);
        tempmeas= *((chromosome[prevent])->gene+(int)roundiat) ;
        tempmeas = tempmeas ^ (unsigned int)c;
        *((chromosome[prevent])->gene+(int)roundiat)=tempmeas;

       } /* matches if...*/
  } /* matches for/next..*/

}
/*********************************************************************/

/*********** CROSSING PARTNER SELECTION FUNCTION
*****************/

void reject(void)
{
 unsigned int *tempop[NUMCHROM]; /* memory handling */
 static int *m1,*m2;
 unsigned int testing,testing2;

 int mate1,mate2,mate1temp,mate2temp; /* Variables concerned with selection of */
 int crossindex;              /* of chromosomes */
 int mflag,hamflag;
 int i,j,numind,chk,memhand,mate1hand,mate2hand;
 int hamthresh,hamdist,tempsize,tempsize2;
 div_t lastchromtemp;                   /* physical crossing of chromosomes */
 unsigned int temp1,temp2;
 unsigned int crossgene1,crossgene2;
 int lastchrom,resub,resubind,subchk;
 int numspeciesmate1,numspeciesmate2,crosspointfinal;
 double round,roundtemp,proby,crosspoint;
 static int alreadydone=0;

 if(!alreadydone)
  {
        if((m1=(int *)malloc(NUMCHROM *sizeof(int)))==NULL)
                {
                printf("problems allocating the first crossing mate \n");
                exit(1);
```

```
        }

      if((m2=(int *)malloc(NUMCHROM *sizeof(int)))==NULL)
        {
               printf("problems allocating the second crossing mate \n");
               exit(1);
        }
        alreadydone=1;
  }

crossindex=0;

 for(i=0;i<(CROSSPOOL);i++)
  {
       mate1temp = pickrand(0,NUMCHROM-1);
       mate1=*(geneindex+mate1temp);
       numind=0;
       do
        {
               mflag =0;
               hamflag=0;
               mate2temp = pickrand(0,NUMCHROM-1);
               mate2=*(geneindex+mate2temp);
               if((chromosome[mate1])->parm != (chromosome[mate2])->parm)
                          mflag =1;

               if(mflag==0)
                      {
                       hamthresh=(chromosome[mate1])->parm;
                       hamdist=hamming(mate1,mate2);
                       if(hamdist<hamthresh)
                              hamflag=1;
                       }

               numind++;
               chk=0;
               if((mflag !=0) || (hamflag !=0))
                       chk=1;
        }
       while((chk ==1) && (numind < (100*CROSSPOOL)));

  if((mflag==0)&&(hamflag==0))
        {
         *(m1+crossindex)=mate1;
         *(m2+crossindex)=mate2;
         crossindex++;
        }

  }/*end of for */
```

```
for(j=0;j<2*crossindex;j+=2)
 {
        testing=*(m1+j/2);
        memhand=(chromosome[testing])->parm;
       /* printf("chromosome size %d\n",memhand);*/
        tempop[j]=(unsigned int *)malloc(memhand*sizeof(unsigned int));
        if(tempop[j]==NULL)
         {
               printf("problems establishing the first cross mate \n");
               exit(1);
               }

        tempop[j+1]=(unsigned int *)malloc(memhand*sizeof(unsigned int));
        if(tempop[j+1]==NULL)
               {
                printf("problems establishing the second mate \n");
                exit(1);
               }

         mate1hand=testing;
         mate2hand=*(m2+j/2);
         crosspoint=0;
         proby = (double)randnew();
         round = proby * ((memhand * WORDSIZE)-1);
         roundtemp = modf(round,&crosspoint);
         if(roundtemp > 0.5)
                      crosspoint = crosspoint +1;

       lastchromtemp = div((int)crosspoint,WORDSIZE);
       lastchrom =lastchromtemp.quot;
       crosspointfinal = lastchromtemp.rem;

       for(i=0;i<lastchrom;i++)
        {
               *(tempop[j]+i)=*((chromosome[mate1hand])->gene+i);
               *(tempop[j+1]+i)=*((chromosome[mate2hand])->gene+i);
        }
      crossgene1=*((chromosome[mate1hand])->gene+lastchrom);
      crossgene2=*((chromosome[mate2hand])->gene+lastchrom);

      cross(crosspointfinal,crossgene1,crossgene2,&temp1,&temp2);


      *(tempop[j]+lastchrom)=temp1;
      *(tempop[j+1]+lastchrom)=temp2;

      for(i=lastchrom+1;i<memhand;i++)
       {
        *(tempop[j]+i)=*((chromosome[mate2hand])->gene+i);
        *(tempop[j+1]+i)=*((chromosome[mate1hand])->gene+i);
```

```
        }

  }

/* resubstitution into original population */

 for(j=0;j<2*crossindex;j+=2)
  {
        testing=*(m1+j/2);
        if(testing==*(geneindex+NUMCHROM-1))
         {
          subchk=0;
          resub=0;
          do
            {
             resubind=*(geneindex+resub);
             if((chromosome[testing])->parm==(chromosome[resubind])->parm)
              {
                 testing=resub;
                 subchk=1;
                 }
             resub++;
            }
           while((resub <NUMCHROM-1)&&(subchk==0));
         }

        testing2=*(m2+j/2);
        if(testing2==*(geneindex+NUMCHROM-1))
         {
          subchk=0;
          resub=0;
          do
            {
            resubind=*(geneindex+resub);
            if((chromosome[testing2])->parm == (chromosome[resubind])->parm)
                  {
                  testing2=resub;
                  subchk=1;
                  }
            resub++;
            }
           while((resub< (NUMCHROM-1))&&(subchk==0));
          }
        memhand=(chromosome[testing])->parm;

         for(i=0;i<memhand;i++)
           {
                 *((chromosome[testing])->gene+i)= *(tempop[j]+i);
                 *((chromosome[testing2])->gene+i)=*(tempop[j+1]+i);
           }
```

```
            free(tempop[j]);
            free(tempop[j+1]);


    } /* end of for-next  */




  }  /* end of function */

/*****************************************************************************
****/


/**********************************************************************/
        /*              GENE CROSSING                        */
        /*  Crossing the bits of the respective genes by converting the */
        /*  binary representation of the denary number into a binary   */
        /*  array (with a bit mask) and performing the sawp at the     */
        /*  bit position on the gene. The resultant genes are          */
        /*  reconverted to denary numbers which are returned to the     */
        /*  calling function by reference via temp1 and temp2.         */

/**********************************************************************/

void cross(int cpoint,unsigned int gene1,unsigned int gene2,unsigned int
*temp1,unsigned int *temp2)
{
 int tempgen1[WORDSIZE],tempgen2[WORDSIZE];
 int gene1st[WORDSIZE],gene2st[WORDSIZE];
 int count;
 double num,conv1,conv2,now;
 unsigned int mask = 0x8000;


 /**** convert number from 0 - 2^16 to its binary string ****/
 for(count=0;count<WORDSIZE;count++)
   {
        gene1st[(WORDSIZE-1)-count] = (mask & gene1) ? 1 : 0;
        gene2st[(WORDSIZE-1) - count] = (mask & gene2) ? 1 : 0;
        mask >>=1;
        }
 /***** copy the binary components of the gene into the ***/
 /***** template genes up to the crossover  point       ***/

 for(count=0;count<cpoint;count++)
```

```
  {
        tempgen1[count] = gene1st[count];
        tempgen2[count] = gene2st[count];
  }

 /**** swap the base pairs at the cross point ***/
 for(count=cpoint;count<WORDSIZE;count ++)
  {
        tempgen1[count] = gene2st[count];
        tempgen2[count] = gene1st[count];
  }

 /***** reconvert the binary strings to their denary  ***/
 /**** equivalent between the range 0 - 2^16          ***/

 conv1=0.0;
 conv2=0.0;
 num = 2.0;

 for(count=0;count<WORDSIZE;count ++)
        {
         now=pow(num,(double)count);
         conv1 = conv1 + (double)tempgen1[count]*now;
         conv2 = conv2 + (double)tempgen2[count]*now;
         }
 *temp1 = (unsigned int) conv1;
 *temp2 = (unsigned int) conv2;

}
         /**** end of gene crossing function *****************/

/**********************************************************************
***/
/*    Hamming distance comparison between two chromosomes            */
/**********************************************************************
***/

int hamming(int chrom1,int chrom2)
{
 int gene1st,gene2st;
 int count,hd,count2;
 unsigned int mask ; /* change this back later to 0x8000 */

 hd=0;
 /**** convert number from 0 - 2^16 to its binary string ****/
 for(count2=0;count2<(chromosome[chrom1])->parm;count2++)
        {
         mask = 0x8000;
         for(count=0;count<WORDSIZE;count++)
        {
```

```
        gene1st = (mask & *((chromosome[chrom1])->gene+count2)) ? 1 : 0;
        gene2st = (mask & *((chromosome[chrom2])->gene+count2)) ? 1 : 0;
        mask >>=1;
        if(gene1st != gene2st)
            hd++;
        }
        }
 return(hd);
}

/**** end of Hamming distance calculation between two similar models ***/


/*************************************************************************
***/
/* DAVIS FUNCTION: Calculates the activity coefficient of a species     */
/* based on its charge and the ionic strength of the system, using the   */
/* Davis 0.2 expression                              */
/*************************************************************************
***/

double davis(double ionicstr,int charge)
{
 double ftemp;
 double base = 10.0;
 ftemp=sqrt(ionicstr)/(sqrt(ionicstr) + 1.0);
 ftemp=ftemp -0.2*ionicstr;
 ftemp=ftemp*charge*charge*(-0.512);
 ftemp=pow(base,ftemp);
 return(ftemp);

}
/****** end of davis expression  ****/




/*********** Start of equil.c program merger **************/
double *equilcalc(double **beta,int chromind,int dataind)
{
  static int alreadydone=0;

  int i,j,k,kind,l,metind,ligind;
  int maxm,maxn,maxv,vtemp;
  static double *vv, *vp, *x, *ki, *logk  ;
  static double *c0, *c ;
  double sum;

  for (i=0; i < NMAX ; i++)         /* initialise the v matrix to zero for */
```

```
  for (j=0; j < MMAX ; j++)        /* elements not involved in a particular */
   v[i][j]=0.0;                    /* equilibrium */

 nequil=(chromosome[chromind])->modord[0]*(chromosome[chromind])->modord[1];
/* number of equilibria */
 mequil= nequil +(chromosome[chromind])->prop[0]+(chromosome[chromind])-
>prop[1];  /* number of species    */
 maxm=MAXSPECIES;
 maxn=MAXEQUIL;
 maxv=maxm*maxn;
 metind=1;
 ligind=1;
 /* Filling The V Matrix */
  for(i=0;i<nequil;i++)
   {
   for(j=0;j<mequil;j++)
     {
       if(i>=(chromosome[chromind])->modord[1])
        {
         if(ligind >(chromosome[chromind])->modord[1])
          {
             ligind=1;
             metind++;
          }
        }

       if(j==0)
         v[i][j]=metind;
       if(j==1)
         v[i][j]=ligind;

       if(metind==1)
        {
         if(j==(v[i][0]*v[i][0] + v[i][1]))
           v[i][j]=-1;
        }
       else
        {
        if((chromosome[chromind])->modord[1] >=1)
          {
          vtemp=v[i][0] +v[i][1] + (chromosome[chromind])->modord[1];
          if( j==(vtemp -1))
               v[i][j] = -1;
          }
        }


     }
    ligind++;
   }
```

```
 if(!alreadydone)
  {
   vv=malloc(maxv*sizeof(double));
   if (vv==NULL) fault(401);
   vp=malloc(maxv*sizeof(double));
  if (vp==NULL) fault(402);
  x=malloc(maxn*sizeof(double));
  if (x==NULL) fault(403);
  ki=malloc(maxn*sizeof(double));
  if (ki==NULL) fault(404);
  logk=malloc(maxn*sizeof(double));
  if (logk==NULL) fault(405);
  c0=malloc(maxm*sizeof(double));
  if (c0==NULL) fault(406);
  c=malloc(maxm*sizeof(double));
  if (c==NULL) fault(407);
  alreadydone=1;
  }

  for(i=0; i< mequil; i++)
     c0[i]=0.0;
 m0=(chromosome[chromind])->prop[0]+(chromosome[chromind])->prop[1];

 c0[0]=*(totmet+dataind);
 c0[1]=*(ligconc+dataind);

/*  printf("\n\nEquil. constants\n\n");    */
 kind=0;
 for(i=1;i<=(chromosome[chromind])->modord[0];i++)
   for(j=1;j<=(chromosome[chromind])->modord[1];j++)
    {
    ki[kind]=*(*(beta+i)+j);
    kind++;
   }
  /* convert from log in the base 10 to log in the base e */

  for(i=0;i<kind;i++)
    logk[i]= ki[i]*log(10.0);
/************************************************/

/* here fill vv matrix
*/

for (i=1; i <= nequil ; i++)
 for (j=1; j <= mequil; j++)
   *elem(vv,mequil,i,j) = *elem(v[0],MMAX,i,j);

guess(nequil,mequil,vv,c0,x,c);

for(i=0;i<nequil*mequil; i++)
```

```
      vp[i]=vv[i];
iter(nequil,mequil, vp,c,x,logk);
 /* printf("\n");*/

/* print results */

/*printf("\nResults\n*******\n\n");
printf("Species                C0              C\n");
printf("----------------------------------------------\n");
for (i=0;i<mequil; i++)
 printf("component %d %15.4e %15.4e\n",i,c0[i],c[i]);  */

return(c);


}

/****** end of main calling routine for the EQUIL.C program ************/

/* DIAG() ROUTINE : returns the value of diagonal element of a square */
/* matrix of doubles of size nequil numbering of elements starts at 1 */

double *diag( double *a, int nequil, int diagnr)
{
  double *diagonal ;
  diagonal= a+ ((diagnr-1)*(nequil+1));
  return (diagonal);
}
/************************************************************************
*****/

/* ELEM() ROUTINE :returns the value of the matrix element of a matrix with */
/* row size nequil and with indices row, column numbering starts at 1       */

double *elem( double *a, int nequil, int row, int col)
{
  double *element ;
  element = a + (nequil*(row-1) + col-1) ;
  return(element);
}
/************************************************************************
*****/

/************** MINA() returns the minimum of two values
*******************/
double mina(double a, double b)
 {
 return( (a >= b) ? b : a);
 }
```

```
/**********************************************************************
******/

/*************** MAXA() returns the maximum of two values
*******************/
double maxa(double a, double b)
 {
  return( (a >= b) ? a : b);
 }
/**********************************************************************
******/

/************** FAULT() prints error message as stated in paper *************/
void fault(int nequil)
 {
   printf("ERROR= %d\n",nequil);
   exit(nequil);
 }
/**********************************************************************
*******/

/********************* DETSYM() ROUTINE START
*******************************/
void detsym( double *a, int nequil, double *minpiv)
{
   double r, s ;
   int k,l,j ;
   for ( k=1; k <= nequil ; k++)
     {
       r=  *diag(a, nequil, k);
       for (l=1; l <= k-1 ; l++)
             r= r - *elem(a, nequil,l,k)*(*elem(a,nequil,l,k));
       if (k==1) *minpiv=r;
       *minpiv=mina(*minpiv,r);
       if ( r <= 0.0) return;
       *diag(a,nequil,k)=r=sqrt(r);
       for (j=k+1; j <= nequil ; j++)
           {
             s=*elem(a,nequil,k,j) ;
             for (l=1; l <= k-1; l++)
               s=s-(*elem(a,nequil,l,j))*(*elem(a,nequil,l,k));
             *elem(a,nequil,k,j)= s/r;
           }
     }
}
/**********************************************************************
****/

/****************** SOLSYM() FUNCTION START
*******************************/
```

```
void solsym(double *a, int nequil, double *b)
{
  double s;
  int i,k ;
  for (i=1; i <= nequil; i++)
  {
    s= b[i-1];
    for (k=1; k <= i-1; k++)
        s=s - *elem(a,nequil, k,i)*b[k-1];
    b[i-1]=s/(*diag(a,nequil,i));
  }
  for (i=nequil; i >= 1; i--)
  {
        s=b[i-1];
        for (k=i+1; k <=nequil ; k++)
          s=s- (*elem(a,nequil,i,k)*b[k-1]);
        b[i-1]=s/(*diag(a,nequil,i));
   }
}
/**********************************************************************
******/

/*********** START OF INITIAL GUESS FOR EQUILIBRIUM
CONCENTRATION ***********/
void guess(int nequil, int mequil, double *v, double *c0 , double *x, double *c)
{
  double c0sum, at, bt, an, bn, a, b ;
  int i, j, first, ready ;
  for (i=0; i < nequil ; i++)
     x[i]=0.0;
  c0sum=0.0 ;
  for (i=0; i < mequil; i++)
  {
   c0sum += c0[i];
   c[i]=c0[i];
  }
LB1: first=0;
   ready=1;

 for(i=1; i <=nequil ; i++)
  {
  if(x[i-1] != 0.0 ) goto LB2;
  at=bt=c0sum;
  an=bn=0.0;
  for(j=1;j<=mequil;j++)
    {
     if(*elem(v,mequil,i,j) > 0.0 )
         {
          at=mina(at,c[j-1]);
          an=maxa(an,*elem(v,mequil,i,j));
```

```
        }
    if(*elem(v,mequil,i,j) < 0.0 )
        {
        bt=mina(bt,c[j-1]);
        bn=maxa(bn,-(*elem(v,mequil,i,j)));
        }
   }

 if((an==0.0) || (bn==0.0)) fault(21) ;
 a=at/an;
 b=bt/bn;
 if((a==0.0) && (b==0.0))
  {
   ready=0;
   goto LB2;
  }

  if((a !=0.0) && (b != 0.0)) goto LB2;

  first=1;
  x[i-1]=0.5*(a-b);
  for(j=1;j<=mequil;j++)
   {
   c[j-1]=c[j-1]-x[i-1]*(*elem(v,mequil,i,j));
   }
  LB2:  continue ;
 }
if (ready==1) return;
if (first==1) goto LB1;
fault(22);
return;
}
/************************************************************************
******/

/* ITER() ROUTINE: to iteratively improve the estimate of the equilibrium   */
/* concentration of the species described in the V matrix */

void iter(int nequil, int mequil, double *v, double *c, double *x, double *logk)
{
   double tolc, epsc, p0, minpiv, fmax0, fmax1, wr, alfa ;
   int i,j, k, it, itmax ;
   static double *w, *z, *f, *dx, *dc, *logc, sum;
   static int alreadydone=0;
   int maxm=10;
   int maxv=50;
   int maxn=5;
 /* here allocations for dynamic array storage */

   if(!alreadydone)
```

```
    {
        w= malloc(maxv*sizeof(double));
        if (w==NULL) fault(201);
        z=malloc(maxv*sizeof(double));
        if (z==NULL) fault(202);
        f=malloc(maxn*sizeof(double));
        if (f==NULL) fault(203);
        dx=malloc(maxn*sizeof(double));
        if (dx==NULL) fault(204);
        dc=malloc(maxm*sizeof(double));
        if (dc==NULL) fault(205);
        logc=malloc(maxm*sizeof(double));
        if (logc==NULL) fault(206);
        alreadydone=1;
    }
  itmax=1000;
  tolc=1.0e-10;
  p0=1.0e-3;
  for (i=1; i <= nequil; i++)
   for (j=1; j <= i ; j++)
        {
         sum = 0.0 ;
          for (k=1; k <= mequil ; k++)
                sum += (*elem(v,mequil,i,k))* (*elem(v,mequil,j,k));
         *elem(z,nequil,i,j)= *elem(z,nequil,j,i)= sum;
        }
  detsym(z,nequil, &minpiv);
  wr= *elem(z,nequil,1,1);
  for (i=2;i <=nequil ; i++)
    wr=wr* (*diag(z,nequil,i));
  if ((minpiv <= 0.0) || (wr < 0.5)) fault(31);
  fmax0=0.0;
  for (j=0; j< mequil ; j++)
    {
/*   printf(" c[%d] is %e \n",j,c[j]);   */
    logc[j]=log(c[j]);
    }
   for (i=1; i <= nequil ; i++)
   {
        sum=0.0;
        for (k=1; k<= mequil; k++)
         sum += *elem(v,mequil,i,k)* logc[k-1];
        f[i-1]=logk[i-1]+sum;
        fmax0=maxa(fmax0, fabs(f[i-1]));
   }
  for (it=1 ; it<= itmax ; it++)
  {
   /* printf(".");*/
    for (j=1; j <= mequil ; j++)
     for (k=1; k<= nequil; k++)
```

```
        *elem(w,mequil,k,j)= *elem(v,mequil,k,j)/ c[j-1];

    for (i=1; i <= nequil ; i++)
    for (j=1; j<= i; j++)
    {
        sum=0.0;
        for (k=1; k<= mequil; k++)
         sum += *elem(w,mequil,i,k)*(*elem(v,mequil,j,k));
         *elem(z,nequil,i,j) = *elem(z,nequil,j,i) = sum;
    }
   detsym(z,nequil,&minpiv);
   if (minpiv <= 0.0) fault(32);
   solsym(z,nequil,f);
   for (i=1; i <= nequil; i++)
         {
           dx[i-1]=f[i-1];
         }
   alfa=1.0;
   epsc=0.0;
   for (j=1; j <=mequil ; j++)
   {
         sum=0.0;
         for (k=1; k<= nequil; k++)
           sum += *elem(v,mequil,k,j)* dx[k-1];
         dc[j-1]=-sum;
/*      printf("dc[%d] is % e\n",j,dc[j-1]); */
         wr=dc[j-1]/c[j-1];
/*      printf("wr is %e epsc is %e \n",wr, epsc); */
         epsc=maxa(epsc,fabs(wr));
         if ( (1.0+ alfa*wr) < p0) alfa=(p0-1.0)/wr;
    }
    if (epsc < tolc )
         {
           return;
          }
LC:   for (j=1; j<=mequil; j++)
         {
          logc[j-1]=log(c[j-1]+alfa*dc[j-1]);
/*        printf("logc[%d] is %e ,alfa is %e \n",j,logc[j-1],alfa); */
         }
   fmax1=0.0;
   for (i=1; i <=nequil ; i++)
   {
         sum=0.0;
         for (k=1;k <=mequil ; k++)
           {
            sum += *elem(v,mequil,i,k)*logc[k-1];
           }
         f[i-1]=logk[i-1]+sum;
         fmax1=maxa(fmax1,fabs(f[i-1]));
```

```
    }
    if ( fmax1 > fmax0 )
        {
         alfa *= 0.5;
         goto LC ;
        }
    fmax0=fmax1;
    for (k=1; k<=nequil; k++)
        x[k-1] += alfa*dx[k-1];
    for (k=1; k <=mequil ; k++)
        c[k-1] += alfa*dc[k-1];
   }
    for (k=1; k<=nequil ; k++)
       printf("%g ",x[k-1]);
    printf("\n");
    fault(33);
    return;
}


/***********************************************************************
***/


/* This is file INDEX.C */
/* This file may have been modified by DJ Delorie (Jan 1991).  If so,
** these modifications are Coyright (C) 1991 DJ Delorie, 24 Kirsten Ave,
** Rochester NH, 03867-2954, USA.
*/

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted provided
 * that: (1) source distributions retain this entire copyright notice and
 * comment, and (2) distributions including binaries display the following
 * acknowledgement:  ``This product includes software developed by the
 * University of California, Berkeley and its contributors'' in the
 * documentation or other materials provided with the distribution and in
 * all advertising materials mentioning features or use of this software.
 * Neither the name of the University nor the names of its contributors may
 * be used to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */
```

```
#if defined(LIBC_SCCS) && !defined(lint)
static char sccsid[] = "@(#)index.c	5.6 (Berkeley) 6/23/90";
#endif /* LIBC_SCCS and not lint */

#include <string.h>
#include <stddef.h>

char *
#ifdef STRCHR
strchr(p, ch)
#else
index(p, ch)
#endif
	register const char *p, ch;
{
	for (;; ++p) {
		if (*p == ch)
			return((char *)p);
		if (!*p)
			return((char *)NULL);
	}
	/* NOTREACHED */
}
```

Gasimp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "randhead.c"
#include "fdist.c"

#define ALPHA 1.0
#define BETA 0.5
#define GAMMA 2.0
#define NUMCHROM 15
#define NUMPARM 15
#define NUMPROP 2
#define MAXMETAL 5

#define RIJ 4
#define KOLOM 4
#define NMAX 20
#define MMAX 20
#define LMAX  10

#define R 8.31441
#define F 96485.309

#define MAXPARM 20
#define MAXSPECIES 10
#define MAXEQUIL 5
#define MAXLIG 5
#define WORDSIZE 16
#define MAXWORD 65535
#define MAXDATA 50
#define CROSSPOOL 2
#define CROSSRATE 0.65
#define NUMELEC 2
#define T 298.15
#define SCONV R*T/(NUMELEC*F)
#define CHARGEMETAL 0
#define CHARGELIGAND 0

void meminit(void);
void modinit(void);
void filhand(void);
void foval(void);
void simplex(void);
double calcfunc(double *pinput);
double *equilcalc(double **beta,int dataind);

void detsym(double *,int nequil,double *);
void solsym(double *, int, double *);
void guess(int, int, double *, double *, double *, double *);
```

```
void iter(int , int, double *, double *, double *, double *);
double *diag( double *, int, int);
double *elem( double *, int, int, int);
double mina(double, double);
double maxa(double, double);
void fault(int);
void readc(void);
void probeer(double *, int, int);
void name(void);
void term(void);
void side(void);
char *index(register const char *,register const char);

struct chromosomedata
  {
   int geneindex;
   int prop[NUMPROP];
   int modord[NUMPROP];
   int parm;
   double *gene;
   double fitness;
   };

struct chromosomedata *chromosome[NUMCHROM];
double maxstab,minstab,averagefitness;
double test,maxheat,minheat;
double *totmet,*ligconc,*Ep,*Ipeak,*Eobs;

int numdata;
int chargemetal=0.0;
int chargeligand=0.0;
char symb[LMAX] ;
int prec ;
int coeff;
int nchar;
char tsymb[MMAX][LMAX];
double v[NMAX][MMAX] ;
int tprec[MMAX], ordec0[MMAX];
int nequil,mequil, m0;
int signum ;


main()
{
 meminit();
 modinit();
 filhand();
 foval();
 simplex();
```

```
}


/**********************************************************/
/*  Memory initialisation for experimental parameters/ fvalues and models   */
/**********************************************************/

void meminit(void)
{
 int i;
 for(i=0;i<NUMCHROM;i++)
   {
    if((chromosome[i]=(struct chromosomedata *)malloc(sizeof(struct
chromosomedata)))==NULL)
     {
       printf("problems establishing the population  structures \n");
       exit(1);
     }
   }

if((ligconc=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
   printf("problems allocating the ligand concentration \n");
   exit(1);
  }

 printf("delatE allocation \n");

 if((totmet=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
   printf("problems establishing the total metal matrix\n");
   exit(1);
  }

/* establishing observed potential shift */
if((Eobs=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
 {
  printf("problems establishing the f value matrix \n");
  exit(1);
 }

if((Ep =(double *)malloc(MAXDATA*sizeof(double)))==NULL)
 {
  printf("problems allocating the change in peak potential \n");
  exit(1);
 }

printf("Ipeak allocation \n");

if((Ipeak=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
```

```
 {
  printf("problems allocating the peak current \n");
  exit(1);
 }

printf("mltit allocation \n");
}

/**************************************************/
/**************************************************/
/*   Initialisation of models for simplex and model parameters  */
/**************************************************/

void modinit(void)
{
 int i,j,metal,ligand,genesize;

 for(i=0;i<NUMCHROM;i++)
  {
  for(j=0;j<2;j++)
     (chromosome[i])->prop[j]=1;

  metal=1;
  (chromosome[i])->modord[0]=metal;
  ligand=3;
  (chromosome[i])->modord[1]=ligand;

   /* considers length of gene sequence with respect to */
   /* stability constants  and heats of formation */
  genesize=ligand*metal; /* +(metal-1) +(ligand-1); self-complexation */

  (chromosome[i])->parm=genesize;
  (chromosome[i])->gene=(double *)malloc(genesize*sizeof(double));
  if((chromosome[i])->gene==NULL)
   {
    printf("not enough memory to initialise the genepool \n");
    exit(1);
   }
  }

 *((chromosome[0])->gene+0)=1.329793;
 *((chromosome[0])->gene+1)=1.72128;
 *((chromosome[0])->gene+2)=1.538842;

 *((chromosome[1])->gene+0)=1.312642;
 *((chromosome[1])->gene+1)=1.812284;
 *((chromosome[1])->gene+2)=1.242634;

 *((chromosome[2])->gene+0)=1.306844;
 *((chromosome[2])->gene+1)=1.787686;
```

```
 *((chromosome[2])->gene+2)=1.397726;


 *((chromosome[3])->gene+0)=1.296468;
 *((chromosome[3])->gene+1)=1.812833;
 *((chromosome[3])->gene+2)=1.333516;


}
/**********************************************************************/

/**************** FILE HANDLING FUNCTION ************************/
void filhand(void)
{
 FILE *fptr;
 int tempcount;
 char comma;
/* double templig[MAXDATA],tempE[MAXDATA],tempi[MAXDATA]; */
/* clrscr();*/
 numdata =0;

 if((fptr=fopen("cadchlor.csv","r"))== NULL)
        {
         printf("sorry cannot open the data file\n");
         exit(1);
        }


while(fscanf(fptr,"%lf%c%lf%c%lf%c%lf",totmet+numdata,&comma,ligconc+numdata,
&comma,Ep+numdata,&comma,Ipeak+numdata) != EOF)
  {
         printf("data
%lf,%lf,%lf,%lf\n",*(totmet+numdata),*(ligconc+numdata),*(Ep+numdata),*(Ipeak+nu
mdata));
         numdata = numdata +1;
        }

 fclose(fptr);

}
/************* end of file handling function *******************************/

/**CALCULATING Fo VALUES FROM THE EXPERIMENTAL DATA ********/
void foval(void)
{
  double tempi,tempe,tempconc,ibase,evolt;

  int j,k;
  k=0;
  ibase= *(Ipeak+k);
```

```
 evolt=*(Ep+k);
 tempconc=*(totmet+k);

 for(j=1;j<numdata;j++)
   {
    tempi=*(totmet+j)/tempconc;
    tempi=tempi*ibase;
    tempi=*(Ipeak+j)/tempi;
    tempi=log(tempi)*SCONV;
    tempe=evolt - *(Ep+j);
    tempe=tempe-tempi;
    *(Eobs+j)=tempe;
   }
}
/************** end of Eobsve determination ************************/

void simplex(void)
{
 /***************************************************************/
 /* Arrays : p[] is the simplex array of size ndim+1,ndim              */
 /*           y[] is a one dimensional array of size ndim              */
 /*           containing the performances of each vertex               */
 /*            beta[] is a two dimensional array containing            */
 /*            the stability constants read from a                     */
 /*            chromosome                                              */
 /***************************************************************/

 double *p[NUMPARM];
 double *pbar,*pr,*pref,*y;
 int i,j,k;
 int ihi,ilo,inhi,iter,flag,flagconv;
 int ndim,betaind,genind;
 double rtol;
 double m,l,ypr,yprr;
 FILE  *fptrsimp;

  int maxconciter=500;
  double convtol= 1e-10;

ndim=3;

 for(i=0;i<ndim+1;i++)
   {
    if((p[i]=(double *)malloc(ndim*sizeof(double)))==NULL)
         {
          printf("problem establishing the vertices for the simplex \n");
          exit(1);
          }
   }
```

```
if((y=(double *)malloc((ndim+1)*sizeof(double)))==NULL)
    {
     printf("problem establishing the performance of each vertex \n");
     exit(1);
     }

  if((pbar=(double *)malloc(ndim*sizeof(double)))==NULL)
     {
      printf("problem establishing the mean point of a face \n");
      exit(1);
      }

  if((pr=(double *)malloc(ndim*sizeof(double)))==NULL)
     {
      printf("problem establishing a reflection point \n");
      exit(1);
      }

  if((pref=(double *)malloc(ndim*sizeof(double)))==NULL)
     {
      printf("problem establishing an extended reflection point \n");
      exit(1);
      }

if((fptrsimp=fopen("simpout.csv","a"))==NULL)
          {
           printf("cannot open the simplex output file \n");
           exit(1);
          }

  for(i=0;i<ndim+1;i++)
    for(j=0;j<ndim;j++)
      *(p[i]+j)=*((chromosome[i])->gene+j);


   for(k=0;k<ndim+1;k++)
     {
     y[k]=calcfunc(p[k]);
     printf("%lf\n",y[k]);
     }

flag=0;
flagconv=0;
iter=0;
 do
 {
   for(k=0;k<ndim+1;k++)
     y[k]=calcfunc(p[k]);

   ilo=0;
```

```
if(y[0] > y[1])
 {
  ihi=0;
  inhi=1;
 }
else
 {
  ihi=1;
  inhi=0;
 }

for(i=0;i<ndim+1;i++)
 {
   if(y[i]< y[ilo])
    ilo=i;

   if(y[i] > y[ihi])
    {
       inhi=ihi;
       ihi=i;
    }
   else if(y[i] > y[inhi])
       if(i != ihi)
         inhi=i;
 }


if(y[ilo]<convtol)
   flagconv=1;
if((flagconv==1) ||(iter > maxconciter))
  flag= 1;

for(j=0;j<ndim;j++)
  *(pbar+j)=0.0;

for(i=0;i<ndim+1;i++)
  if(i!=ihi)
   for(j=0;j<ndim;j++)
       *(pbar+j)=*(pbar+j) + *(p[i]+j) ;

for(j=0;j<ndim;j++)
  {
   *(pbar+j)=*(pbar+j)/ndim;
   *(pr+j)= *(pbar+j)*(1.0 + ALPHA) - *(p[ihi]+j)*ALPHA;
   if(*(pr+j) >18.0)
       *(pr+j)= 18.0;
  }
```

```
ypr=calcfunc(pr);

if(ypr < y[ilo])
 {
  for(j=0;j<ndim;j++)
   {
    *(pref+j) = *(pr+j)*GAMMA + *(pbar +j)*(1.0-GAMMA);
    if(*(pref+j) >18.0)
        *(pref+j) = 18.0;
    }
   yprr=calcfunc(pref);

   if(yprr < y[ilo])
    {
        for(j=0;j<ndim;j++)
          *(p[ihi]+j) = *(pref+j);
        y[ihi]=yprr;
    }
   else
    {
        for(j=0;j<ndim;j++)
          *(p[ihi]+j)=*(pr +j);
        y[ihi]=ypr;
    }

 }
else if(ypr >= y[inhi])
 {
  if(ypr < y[ihi])
   {
    for(j=0;j<ndim;j++)
        *(p[ihi]+j) = *(pr +j);
    y[ihi]= ypr;
   }

  for(j=0;j<ndim;j++)
   {
    *(pref+j)= *(p[ihi]+j)* BETA + *(pbar +j)*(1.0-BETA);
    if(*(pref+j) >18.0)
         *(pref+j) =18.0;
   }

  yprr= calcfunc(pref);
  if(yprr < y[ihi])
   {
    for(j=0;j<ndim;j++)
        *(p[ihi]+j) = *(pref +j);
    y[ihi]=yprr;
   }
  else
```

```
    {
     for(i=0;i<ndim+1;i++)
         {
          if(i !=ilo)
           {
            for(j=0;j<ndim;j++)
             {
              *(pr+j)= *(p[i]+j) + *(p[ilo]+j);
              *(pr+j)= *(pr+j)*0.5;
              *(p[i]+j)=*(pr+j);
             }
            y[i]=calcfunc(pr);
           }   /* end of if i */
         } /* end for i */
    }  /* end of else */
   } /* end of else if */
  else
   {
    for(j=0;j<ndim;j++)
      *(p[ihi] +j)= *(pr +j);
    y[ihi] =ypr;
    }

  printf("iteration %d error %e\n",iter,y[ilo]);

fprintf(fptrsimp,"%d,%e,%e,%e,%e\n",iter,y[ilo],*(p[ilo]+0),*(p[ilo]+1),*(p[ilo]+2),*(p[
ilo]+3));
  iter++;
} while(flag==0);/* end of iterations */

fclose(fptrsimp);

}   /* end of function */

/*******************************************************************/
/*** DETERMINATION OF MSE FOR A GIVEN CHROMOSOME  *********/
double calcfunc(double *pinput)
{
 static double *beta[MAXMETAL];
 double *concarr;
 double actcoef[MAXSPECIES];
 int betasize,i,betaind,numspecies,tempactpos;
 int j,metalsize,ligsize,tailend,geneptr,enthalpyindex;
 int metind,ligind,datacounter;
 int charge[MAXSPECIES];
 int poleqind=1;
 static int alreadydone=0;

 double error,metal,sum,calce,expte,metalconc,temperror,temphold,thermconv;
 double tempionic,ionicstrength,tempact,tempact2;
```

```
double base= 10.0;
double metalactcoef,ligactcoef;
double ionicstr2,errionic,newtitvolume,vesselvolume;

if(!alreadydone)
  {
   for(i=0;i<=(chromosome[0])->modord[0];i++)    /*self-complexation */
    {
     betasize=(chromosome[0])->modord[1] +1;
     if((beta[i]=(double *)malloc(betasize*sizeof(double)))==NULL)
      {
         printf("problems establishing the beta array \n");
         exit(1);
      }
    }
   alreadydone=1;
  }

 for(i=0;i<=(chromosome[0])->modord[0];i++)
  for(j=0;j<(chromosome[0])->modord[1]+1;j++)
    *(beta[i]+j)=1.0;

 *(beta[0]+0)=0.0;  /*nothing present for complexation */

 for(i=0;i<=(chromosome[0])->modord[0];i++)
  for(j=0;j<(chromosome[0])->modord[1]+1;j++)
   *(beta[i]+j)=1.0;

 *(beta[0]+0)=0.0;  /*nothing present for complexation */
 numspecies=(chromosome[0])->modord[0] * (chromosome[0])->modord[1];
 numspecies= numspecies + (chromosome[0])->prop[0] + (chromosome[0])->prop[1];
  error=0.0;
  vesselvolume=99.99;

 metalsize=(chromosome[0])->modord[0];
 ligsize=(chromosome[0])->modord[1];
 tailend=metalsize*ligsize;
 metind=1;
 ligind=1;

for(i=0;i<tailend;i++)
 {
   if(ligind>(chromosome[0])->modord[1])
     {
        ligind=1;
        metind++;
     }
   temphold=*(pinput+i);
   *(beta[metind]+ligind)=temphold;
   ligind++;
```

```
 }


/* omit the section for self-complexation of the ligand and/or metal */

  error=0;

 for(i=1;i<numdata;i++)
 {
  metalconc=*(totmet+i);
  concarr= equilcalc(beta,i);
  metal=*(concarr+0);
  sum=metalconc/metal;
  calce=log(sum)*SCONV;
  expte=*(Eobs+i);
  error=error+(expte-calce)*(expte-calce);
  }

 return(error);



  for(i=0;i<=(chromosome[0])->modord[0];i++)
     free(beta[i]);
}
/*****************************************************/

/*********** Start of equil.c program merger **************/
double *equilcalc(double **beta,int dataind)
{
 static int alreadydone=0;

 int i,j,k,kind,l,metind,ligind;
 int maxm,maxn,maxv,vtemp;
 static double *vv, *vp, *x, *ki, *logk  ;
 static double *c0, *c ;
 double sum;

 for (i=0; i < NMAX ; i++)           /* initialise the v matrix to zero for */
  for (j=0; j < MMAX ; j++)          /* elements not involved in a particular */
   v[i][j]=0.0;                     /* equilibrium */


 maxm=MAXSPECIES;
 maxn=MAXEQUIL;
 maxv=maxm*maxn;
 metind=1;
 ligind=1;
```

```
nequil=(chromosome[0])->modord[0]*(chromosome[0])->modord[1];  /* number of
equilibria */
mequil= nequil +(chromosome[0])->prop[0]+(chromosome[0])->prop[1];  /* number
of species    */



for(i=0;i<nequil;i++)
 {
 for(j=0;j<mequil;j++)
  {
    if(i>=(chromosome[0])->modord[1])
     {
      if(ligind >(chromosome[0])->modord[1])
       {
         ligind=1;
         metind++;
       }
     }

    if(j==0)
      v[i][j]=metind;
    if(j==1)
      v[i][j]=ligind;

    if(metind==1)
     {
      if(j==(v[i][0]*v[i][0] + v[i][1]))
        v[i][j]=-1;
     }
    else
     {
      if((chromosome[0])->modord[1] >=1)
       {
        vtemp=v[i][0] +v[i][1] + (chromosome[0])->modord[1];
        if( j==(vtemp -1))
            v[i][j] = -1;
       }
     }


  }
 ligind++;
 }

if(!alreadydone)
 {
  vv=malloc(maxv*sizeof(double));
  if (vv==NULL) fault(401);
  vp=malloc(maxv*sizeof(double));
```

```
  if (vp==NULL) fault(402);
  x=malloc(maxn*sizeof(double));
  if (x==NULL) fault(403);
  ki=malloc(maxn*sizeof(double));
  if (ki==NULL) fault(404);
  logk=malloc(maxn*sizeof(double));
  if (logk==NULL) fault(405);
  c0=malloc(maxm*sizeof(double));
  if (c0==NULL) fault(406);
  c=malloc(maxm*sizeof(double));
  if (c==NULL) fault(407);
  alreadydone=1;
  }

  for(i=0; i< mequil; i++)
     c0[i]=0.0;
 m0=(chromosome[0])->prop[0]+(chromosome[0])->prop[1];

 c0[0]=*(totmet+dataind);
 c0[1]=*(ligconc+dataind);

/*  printf("\n\nEquil. constants\n\n");    */
 kind=0;
 for(i=1;i<=(chromosome[0])->modord[0];i++)
   for(j=1;j<=(chromosome[0])->modord[1];j++)
    {
     ki[kind]=*(*(beta+i)+j);
     kind++;
    }
  /* convert from log in the base 10 to log in the base e */

  for(i=0;i<kind;i++)
    logk[i]= ki[i]*log(10.0);
/**************************************************/

/* here fill vv matrix
*/

for (i=1; i <= nequil ; i++)
 for (j=1; j <= mequil; j++)
   *elem(vv,mequil,i,j) = *elem(v[0],MMAX,i,j);

guess(nequil,mequil,vv,c0,x,c);

for(i=0;i<nequil*mequil; i++)
      vp[i]=vv[i];
iter(nequil,mequil, vp,c,x,logk);
 /* printf("\n");*/

/* print results */
```

```
/*printf("\nResults\n*******\n\n");
printf("Species              C0              C\n");
printf("---------------------------------------------\n");
for (i=0;i<mequil; i++)
 printf("component %d %15.4e %15.4e\n",i,c0[i],c[i]);  */

return(c);


}

/****** end of main calling routine for the EQUIL.C program ************/

/* DIAG() ROUTINE : returns the value of diagonal element of a square */
/* matrix of doubles of size nequil numbering of elements starts at 1 */

double *diag( double *a, int nequil, int diagnr)
{
  double *diagonal ;
  diagonal= a+ ((diagnr-1)*(nequil+1));
  return (diagonal);
}
/**********************************************************************
*/

/* ELEM() ROUTINE :returns the value of the matrix element of a matrix with */
/* row size nequil and with indices row, column numbering starts at 1      */

double *elem( double *a, int nequil, int row, int col)
{
  double *element ;
  element = a + (nequil*(row-1) + col-1) ;
  return(element);
}
/*********************************************************************/

/************** MINA() returns the minimum of two values ******************/
double mina(double a, double b)
 {
 return( (a >= b) ? b : a);
 }

/*********************************************************************/

/*************** MAXA() returns the maximum of two values ***************/
double maxa(double a, double b)
 {
  return( (a >= b) ? a : b);
 }
```

```
/*********************************************************************/

/************** FAULT() prints error message as stated in paper *************/
void fault(int nequil)
 {
  printf("ERROR= %d\n",nequil);
  exit(nequil);
 }
/*********************************************************************/

/************** DETSYM() ROUTINE START ***************************/
void detsym( double *a, int nequil, double *minpiv)
{
  double r, s ;
  int k,l,j ;
  for ( k=1; k <= nequil ; k++)
   {
    r=  *diag(a, nequil, k);
    for (l=1; l <= k-1 ; l++)
         r= r - *elem(a, nequil,l,k)*(*elem(a,nequil,l,k));
    if (k==1) *minpiv=r;
    *minpiv=mina(*minpiv,r);
    if ( r <= 0.0) return;
    *diag(a,nequil,k)=r=sqrt(r);
    for (j=k+1; j <= nequil ; j++)
        {
         s=*elem(a,nequil,k,j) ;
         for (l=1; l <= k-1; l++)
           s=s-(*elem(a,nequil,l,j))*(*elem(a,nequil,l,k));
         *elem(a,nequil,k,j)= s/r;
        }
    }
}
/*********************************************************************/

/************ SOLSYM() FUNCTION START ****************************/
void solsym(double *a, int nequil, double *b)
{
  double s;
  int i,k ;
  for (i=1; i <= nequil; i++)
  {
    s= b[i-1];
    for (k=1; k <= i-1; k++)
         s=s - *elem(a,nequil, k,i)*b[k-1];
    b[i-1]=s/(*diag(a,nequil,i));
  }
  for (i=nequil; i >= 1; i--)
  {
        s=b[i-1];
```

```
            for (k=i+1; k <=nequil ; k++)
              s=s- (*elem(a,nequil,i,k)*b[k-1]);
            b[i-1]=s/(*diag(a,nequil,i));
    }
}
/**********************************************************************/

/***** START OF INITIAL GUESS FOR EQUILIBRIUM CONCENTRATION ****/
void guess(int nequil, int mequil, double *v, double *c0 , double *x, double *c)
{
  double c0sum, at, bt, an, bn, a, b ;
  int i, j, first, ready ;
  for (i=0; i < nequil ; i++)
     x[i]=0.0;
  c0sum=0.0 ;
  for (i=0; i < mequil; i++)
  {
   c0sum += c0[i];
   c[i]=c0[i];
  }
LB1: first=0;
    ready=1;

 for(i=1; i <=nequil ; i++)
  {
  if(x[i-1] != 0.0 ) goto LB2;
  at=bt=c0sum;
  an=bn=0.0;
  for(j=1;j<=mequil;j++)
    {
     if(*elem(v,mequil,i,j) > 0.0 )
         {
          at=mina(at,c[j-1]);
          an=maxa(an,*elem(v,mequil,i,j));
          }
     if(*elem(v,mequil,i,j) < 0.0 )
          {
           bt=mina(bt,c[j-1]);
           bn=maxa(bn,-(*elem(v,mequil,i,j)));
          }
    }

 if((an==0.0) || (bn==0.0)) fault(21) ;
 a=at/an;
 b=bt/bn;
 if((a==0.0) && (b==0.0))
   {
    ready=0;
    goto LB2;
   }
```

```
  if((a !=0.0) && (b != 0.0)) goto LB2;

  first=1;
  x[i-1]=0.5*(a-b);
  for(j=1;j<=mequil;j++)
   {
    c[j-1]=c[j-1]-x[i-1]*(*elem(v,mequil,i,j));
   }
  LB2:  continue ;
 }
if (ready==1) return;
if (first==1) goto LB1;
fault(22);
return;
}
/*************************************************************************/

/* ITER() ROUTINE: to iteratively improve the estimate of the equilibrium   */
/* concentration of the species described in the V matrix */

void iter(int nequil, int mequil, double *v, double *c, double *x, double *logk)
{
   double tolc, epsc, p0, minpiv, fmax0, fmax1, wr, alfa ;
   int i,j, k, it, itmax ;
   static double *w, *z, *f, *dx, *dc, *logc, sum;
   static int alreadydone=0;
   int maxm=10;
   int maxv=50;
   int maxn=5;
 /* here allocations for dynamic array storage */

   if(!alreadydone)
     {
        w= malloc(maxv*sizeof(double));
        if (w==NULL) fault(201);
        z=malloc(maxv*sizeof(double));
        if (z==NULL) fault(202);
        f=malloc(maxn*sizeof(double));
        if (f==NULL) fault(203);
        dx=malloc(maxn*sizeof(double));
        if (dx==NULL) fault(204);
        dc=malloc(maxm*sizeof(double));
        if (dc==NULL) fault(205);
        logc=malloc(maxm*sizeof(double));
        if (logc==NULL) fault(206);
        alreadydone=1;
     }
   itmax=1000;
   tolc=1.0e-10;
```

```
p0=1.0e-3;
for (i=1; i <= nequil; i++)
 for (j=1; j <= i ; j++)
      {
       sum = 0.0 ;
         for (k=1; k <= mequil ; k++)
                sum += (*elem(v,mequil,i,k))* (*elem(v,mequil,j,k));
       *elem(z,nequil,i,j)= *elem(z,nequil,j,i)= sum;
      }
detsym(z,nequil, &minpiv);
wr= *elem(z,nequil,1,1);
for (i=2;i <=nequil ; i++)
  wr=wr* (*diag(z,nequil,i));
if ((minpiv <= 0.0) || (wr < 0.5)) fault(31);
fmax0=0.0;
for (j=0; j< mequil ; j++)
 {
/*   printf(" c[%d] is %e \n",j,c[j]);   */
  logc[j]=log(c[j]);
 }
 for (i=1; i <= nequil ; i++)
 {
     sum=0.0;
     for (k=1; k<= mequil; k++)
      sum += *elem(v,mequil,i,k)* logc[k-1];
     f[i-1]=logk[i-1]+sum;
     fmax0=maxa(fmax0, fabs(f[i-1]));
 }
for (it=1 ; it<= itmax ; it++)
{
 /* printf(".");*/
  for (j=1; j <= mequil ; j++)
   for (k=1; k<= nequil; k++)
      *elem(w,mequil,k,j)= *elem(v,mequil,k,j)/ c[j-1];

   for (i=1; i <= nequil ; i++)
   for (j=1; j<= i; j++)
   {
      sum=0.0;
      for (k=1; k<= mequil; k++)
       sum += *elem(w,mequil,i,k)*(*elem(v,mequil,j,k));
       *elem(z,nequil,i,j) = *elem(z,nequil,j,i) = sum;
   }
  detsym(z,nequil,&minpiv);
  if (minpiv <= 0.0) fault(32);
  solsym(z,nequil,f);
  for (i=1; i <= nequil; i++)
       {
         dx[i-1]=f[i-1];
       }
```

```
        alfa=1.0;
        epsc=0.0;
        for (j=1; j <=mequil ; j++)
        {
            sum=0.0;
            for (k=1; k<= nequil; k++)
              sum += *elem(v,mequil,k,j)* dx[k-1];
            dc[j-1]=-sum;
/*          printf("dc[%d] is % e\n",j,dc[j-1]); */
              wr=dc[j-1]/c[j-1];
/*          printf("wr is %e epsc is %e \n",wr, epsc); */
              epsc=maxa(epsc,fabs(wr));
              if ( (1.0+ alfa*wr) < p0) alfa=(p0-1.0)/wr;
        }
        if (epsc < tolc )
            {
              return;
            }
LC:   for (j=1; j<=mequil; j++)
            {
              logc[j-1]=log(c[j-1]+alfa*dc[j-1]);
/*            printf("logc[%d] is %e ,alfa is %e \n",j,logc[j-1],alfa); */
            }
        fmax1=0.0;
        for (i=1; i <=nequil ; i++)
        {
            sum=0.0;
            for (k=1;k <=mequil ; k++)
              {
               sum += *elem(v,mequil,i,k)*logc[k-1];
              }
            f[i-1]=logk[i-1]+sum;
            fmax1=maxa(fmax1,fabs(f[i-1]));
        }
        if ( fmax1 > fmax0 )
            {
              alfa *= 0.5;
              goto LC ;
            }
        fmax0=fmax1;
        for (k=1; k<=nequil; k++)
            x[k-1] += alfa*dx[k-1];
        for (k=1; k <=mequil ; k++)
            c[k-1] += alfa*dc[k-1];
        }
        for (k=1; k<=nequil ; k++)
            printf("%g ",x[k-1]);
        printf("\n");
        fault(33);
        return;
```

```
}

/*************************************************************************/

/* This is file INDEX.C */
/* This file may have been modified by DJ Delorie (Jan 1991).  If so,
** these modifications are Coyright (C) 1991 DJ Delorie, 24 Kirsten Ave,
** Rochester NH, 03867-2954, USA.
*/

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted provided
 * that: (1) source distributions retain this entire copyright notice and
 * comment, and (2) distributions including binaries display the following
 * acknowledgement:  ``This product includes software developed by the
 * University of California, Berkeley and its contributors'' in the
 * documentation or other materials provided with the distribution and in
 * all advertising materials mentioning features or use of this software.
 * Neither the name of the University nor the names of its contributors may
 * be used to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#if defined(LIBC_SCCS) && !defined(lint)
static char sccsid[] = "@(#)index.c	5.6 (Berkeley) 6/23/90";
#endif /* LIBC_SCCS and not lint */

#include <string.h>
#include <stddef.h>

char *
#ifdef STRCHR
strchr(p, ch)
#else
index(p, ch)
#endif
	register const char *p, ch;
{
	for (;; ++p) {
		if (*p == ch)
			return((char *)p);
```

```
		if (!*p)
			return((char *)NULL);
	}
	/* NOTREACHED */
}
```

Stdev.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define RIJ 4
#define KOLOM 4
#define NMAX 20
#define MMAX 20
#define LMAX  10

#define CON 1.4
#define CON2 (CON*CON)
#define BIG 1.0e30
#define NTAB 10
#define SAFE 2.0
#define R 8.31441
#define F 96485.309
#define MAXDATA 20
#define MAXMETAL 3
#define MAXPARM 10
#define MAXSPECIES 10
#define MAXEQUIL 10

#define NUMELEC 2
#define T 298.15
#define SCONV R*T/(NUMELEC*F)

struct modeldata
 {
 int metalorder;
 int ligandorder;
 int parm;
 double *string;
 };


 struct modeldata *model[20];

double dfridir(int datapt,int parmindex,double h,double *err);
double FMAX(double a,double b);
void meminit(void);
void modelinit(void);
void filhand(void);
double calcfunc(int filepoint,int parmnum,double parmvalue);

double *diag( double *a, int nequil, int diagnr);
double *elem( double *a, int nequil, int row, int col);
double mina(double a, double b);
double maxa(double a, double b);
void fault(int nequil);
```

```
void detsym( double *a, int nequil, double *minpiv);
void solsym(double *a, int nequil, double *b);
void guess(int nequil, int mequil, double *v, double *c0 , double *x, double *c);
void iter(int nequil, int mequil, double *v, double *c, double *x, double *logk);
double equilcalc(double **beta,int dataind);

void svdcmp(double **a,int m,int n,double *w,double **svd);
double pythag(double a, double b);

double SIGN(double a, double b);
int IMIN(int a,int b);

double *totmet,*ligconc,*Ep,*Ipeak,*Eobs;
double *jacob[MAXDATA],*jacobtrans[MAXDATA],*hessian[MAXDATA],fit1[20];
int numdata,filsize;
double v[NMAX][MMAX] ;
int tprec[MMAX], ordec0[MMAX];
int nequil,mequil, m0;
int signum ,fileloop;

void main(void)
{
 double deriv,x,h,err,try,rss;
 int modloop,dataloop,mulpinloop,mulpoutloop,degfreed;
 int innerloop,outerloop,outerloop2,i,j;
 double w[20],winv[20];
 double *inputa[20],*inputv[20];
 double vsvd[20][20],vtrans[20][20];
 double utrans[20][20],temp1[20][20],temp2[20][20];
 double inverse[20][20];

 double innerprod[20][20],orig[20][20];
 double hessiantemp[20][20];
 FILE *fptrout;

  if((fptrout=fopen("cadout3.csv","w"))==NULL)
    {
     printf("cannot open the output log file \n");
     exit(1);
    }

h=0.2;

meminit();
  modelinit();
  filhand();

    rss=0.000084;
    err=0;
    for(i=0;i<20;i++)
```

```
  for(j=0;j<20;j++)
      vsvd[i][j]=0;

 for(modloop=0;modloop<numdata;modloop++)
  for(dataloop=0;dataloop<numdata;dataloop++)
       hessian[modloop][dataloop]=0.0;

for(modloop=0;modloop<numdata-1;modloop++)
  for(dataloop=0;dataloop<(model[fileloop])->parm;dataloop++)
  {
      deriv=dfridir(modloop+1,dataloop,h,&err);
      jacob[modloop][dataloop]=deriv;
  }


for(modloop=0;modloop<numdata-1;modloop++)
 for(dataloop=0;dataloop<(model[fileloop])->parm;dataloop++)
 jacobtrans[dataloop][modloop]=jacob[modloop][dataloop];

outerloop=(model[0])->parm;
innerloop=(model[0])->parm;
outerloop2=numdata-1;

for(modloop=0;modloop<outerloop;modloop++)
 {
 for(dataloop=0;dataloop<innerloop;dataloop++)
  {
   try=0;
   for(mulpinloop=0;mulpinloop<outerloop2;mulpinloop++)
        try=try+jacobtrans[modloop][mulpinloop]*jacob[mulpinloop][dataloop];
   hessian[modloop][dataloop]=try;
   printf("%lf ",hessian[modloop][dataloop]);
  }
 printf("\n");
 }

for(i=1;i<=(model[0])->parm;i++)
 for(j=1;j<=(model[0])->parm;j++)
   hessiantemp[i][j]=hessian[i-1][j-1];

for(i=0;i<20;i++)
 {
 w[i]=0.0;
 inputa[i]=hessiantemp[i];
 inputv[i]=vsvd[i];
 }

svdcmp(&inputa[0],(model[0])->parm,(model[0])->parm,&w[0],&inputv[0]);
printf("orignial matrix \n");
```

```
for(i=1;i<=(model[0])->parm;i++)
  for(j=0;j<=(model[0])->parm;j++)
      vtrans[i][j]=vsvd[j][i];

for(i=1;i<=(model[0])->parm;i++)
   for(j=1;j<=(model[0])->parm;j++)
     innerprod[i][j]=w[i]*vtrans[i][j];

innerloop=(model[0])->parm;
outerloop=(model[0])->parm;
outerloop2=(model[0])->parm;

for(i=1;i<=outerloop;i++)
  {
   for(j=1;j<=innerloop;j++)
       {
        try=0.0;
        for(mulpinloop=1;mulpinloop<=outerloop2;mulpinloop++)
          try=try+hessiantemp[i][mulpinloop]*innerprod[mulpinloop][j];
        orig[i][j]=try;
       printf("%lf ",orig[i][j]);
       }
   printf("\n");
  }

printf("inverse matrix \n");

for(i=1;i<=(model[0])->parm;i++)
  for(j=0;j<=(model[0])->parm;j++)
       utrans[i][j]=hessiantemp[j][i];

for(i=1;i<=(model[0])->parm;i++)
   winv[i]= 1.0/w[i];


for(i=1;i<=(model[0])->parm;i++)
  for(j=1;j<=(model[0])->parm;j++)
    innerprod[i][j]=winv[i]*utrans[i][j];

innerloop=(model[0])->parm;
outerloop=(model[0])->parm;
outerloop2=(model[0])->parm;
for(i=1;i<=outerloop;i++)
  {
  for(j=1;j<=innerloop;j++)
   {
    try=0.0;
       for(mulpinloop=1;mulpinloop<=outerloop2;mulpinloop++)
           try=try+vsvd[i][mulpinloop]*innerprod[mulpinloop][j];
    inverse[i][j]=try;
```

```
      printf("%lf ",inverse[i][j]);
     }
    printf("\n");
   }

  printf("\n\n");
  innerloop=(model[0])->parm;
  outerloop=(model[0])->parm;
  outerloop2=(model[0])->parm;

  for(i=1;i<=outerloop;i++)
 {
  for(j=1;j<=innerloop;j++)
   {
    try=0.0;
     for(mulpinloop=1;mulpinloop<=outerloop2;mulpinloop++)
      try=try+inverse[i][mulpinloop]*orig[mulpinloop][j];
    temp1[i][j]=try;
    printf("%lf ",temp1[i][j]);
    }
  printf("\n");
  }

/* innerloop=(model[0])->parm;
 outerloop=(model[0])->parm;
 outerloop2=(model[0])->parm;
 for(i=1;i<=outerloop;i++)
 {
  for(j=1;j<=innerloop;j++)
   {
    try=0.0;
     for(mulpinloop=1;mulpinloop<=outerloop2;mulpinloop++)
      try=try+orig[i][mulpinloop]*temp1[mulpinloop][j];
    temp2[i][j]=try;
    printf("%lf ",temp2[i][j]);
    }
  printf("\n");
  }*/
 degfreed=numdata-(model[0])->parm -1.0;
 printf("\n");
 for(i=1;i<=outerloop;i++)
 {
  for(j=1;j<=innerloop;j++)
     temp2[i][j]=inverse[i][j]*rss/degfreed;
  temp2[i][i]=sqrt(temp2[i][i]);
  printf("beta %d value %lf error %lf\n",i,*((model[0])->string+i-1),temp2[i][i]);
  fprintf(fptrout,"%lf, %lf",*((model[0])->string+i-1),temp2[i][i]);
 }
 fprintf(fptrout,"\n");
 printf("\n\n");
```

```
 for(i=1;i<=outerloop;i++)
  {
   for(j=1;j<=innerloop;j++)
     {
      temp2[i][j]=inverse[i][j]/sqrt(inverse[i][i]);
      temp2[i][j]=temp2[i][j]/sqrt(inverse[j][j]);
      printf("%lf  ",temp2[i][j]);
      fprintf(fptrout,"%lf",temp2[i][j]);
     }
  printf("\n");
  fprintf(fptrout,"\n");
  }
/*  getch();    */

 fclose(fptrout);
/* getch();   */

}


/*********************************************************************/
void meminit(void)
{
 int i;
 for(i=0;i<MAXDATA;i++)
  if((jacob[i]=(double *)malloc(MAXPARM*sizeof(double)))==NULL)
   {
    printf("problems establishing the Jacobian matrix \n");
    exit(1);
   }

 for(i=0;i<MAXDATA;i++)
 if((jacobtrans[i]=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
   {
    printf("problems establishing the transpose Jacobian matrix \n");
    exit(1);
   }

 for(i=0;i<MAXDATA;i++)
 if((hessian[i]=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
   {
    printf("problems establishing the Hessian matrix \n");
    exit(1);
   }


 if((Eobs=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
   {
    printf("problems establishing the f value matrix \n");
    exit(1);
```

```
  }

 printf("ligconc allocation \n");

 if((ligconc=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
   printf("problems allocating the ligand concentration \n");
   exit(1);
  }

 printf("delatE allocation \n");

 if((totmet=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
 {
  printf("problems establishing the total metal matrix\n");
  exit(1);
 }

 if((Ep =(double *)malloc(MAXDATA*sizeof(double)))==NULL)
   {
    printf("problems allocating the change in peak potential \n");
    exit(1);
   }

 printf("Ipeak allocation \n");
 if((Ipeak=(double *)malloc(MAXDATA*sizeof(double)))==NULL)
  {
    printf("problems allocating the peak current \n");
    exit(1);
  }

}

/************************************************************/
void modelinit(void)
 {
 int i,size,iternum;
 FILE *fptrin;
 char comma;
 double temp1,temp2,temp3,temp4;
 double avfit;

  if((model[0]=(struct modeldata  *)malloc(sizeof(struct modeldata)))==NULL)
   {
    printf("problems establishing the population  structures \n");
    exit(1);
    }

filsize=0;
```

```
  (model[filsize])->metalorder=2;
   (model[filsize])->ligandorder=1;
   (model[filsize])->parm=2;
   size=(model[filsize])->parm;

   (model[filsize])->string=(double *)malloc(size*sizeof(double));
   if((model[filsize])->string==NULL)
    {
     printf("not enough memory for model string \n");
     exit(1);
    }

  *((model[filsize])->string+0)= 7.459518;
  *((model[filsize])->string+1)= 14.14675;


 }

double FMAX(double a,double b)
{
 double maxarg1,maxarg2;
 maxarg1=a;
 maxarg2=b;
 if(maxarg1 > maxarg2)
   return(maxarg1) ;
 else
  return(maxarg2);

}

/*********************************************************************/
double dfridir(int datapt,int parmindex,double h,double *err)
{
 int i,j;
 double errt,fac,hh,ans;
 double a[NTAB][NTAB];
 double x= *((model[fileloop])->string+parmindex);
 if(h==0.0)
  {
  printf("h must be nonzero for numerical differentiation \n");
  exit(1);
  }

 hh=h;
 *err=BIG;
 a[1][1]=(calcfunc(datapt,parmindex,(x+hh))-calcfunc(datapt,parmindex,(x-
hh)))/(2.0*hh);


 for(i=2;i<=NTAB;i++)
```

```
 {
  hh = hh/CON;
  a[1][i]=(calcfunc(datapt,parmindex,(x+hh)) -calcfunc(datapt,parmindex,(x-
hh)))/(2.0*hh);
  fac=CON2;
  for(j=2;j<=i;j++)
    {
     a[j][i]=(a[j-1][i]*fac -a[j-1][i-1])/(fac -1.0);
     fac=CON2*fac;
     errt=FMAX(fabs(a[j][i]-a[j-1][i]),fabs(a[j][i]-a[j-1][i-1]));
     if(errt <=*err)
         {
          *err=errt;
          ans=a[j][i];
         }
    }/* matches end of for j */

   if(fabs(a[i][i] - a[i-1][i-1]) >= SAFE*(*err))
    {
     printf("higher order is worse by factor SAFE \n");
     break;
    }
  } /* end of for i */
 return(ans);



 }

/*********************************************************************/

/************* FILE HANDLING FUNCTION ****************************/
void filhand(void)
{
 FILE *fptr;
 int tempcount;
 char comma;
/* double templig[MAXDATA],tempE[MAXDATA],tempi[MAXDATA]; */
/* clrscr();*/
 numdata =0;

 if((fptr=fopen("lead.csv","r"))== NULL)
   {
    printf("sorry cannot open the data file\n");
    exit(1);
    }
```

```
while(fscanf(fptr,"%lf%c%lf%c%lf%c%lf",totmet+numdata,&comma,ligconc+numdata,
&comma,Ep+numdata,&comma,Ipeak+numdata) != EOF)
  {
   printf("data
%lf,%lf,%lf,%lf\n",*(totmet+numdata),*(ligconc+numdata),*(Ep+numdata),*(Ipeak+nu
mdata));
   numdata = numdata +1;
  }

 fclose(fptr);

}
/************* end of file handling function *********************************/


/***** DETERMINATION OF MSE FOR A GIVEN CHROMOSOME  *********/
double calcfunc(int filepoint,int parmnum,double parmvalue)
{

 static double *beta[MAXMETAL];
 int betasize,i,betaind;
 int j,metalsize,ligsize,tailend,geneptr;
 int metind,ligind;
 double metal,sum,calce,expte,metalconc;
 double temphold;

 static int alreadydone=0;
 if(!alreadydone)
  {
   for(i=0;i<=(model[fileloop])->metalorder;i++)   /*self-complexation */
    {
     betasize=(model[fileloop])->ligandorder +1;
     if((beta[i]=(double *)malloc(betasize*sizeof(double)))==NULL)
        {
         printf("problems establishing the beta array \n");
         exit(1);
     }
    }
  alreadydone=1;
  }


  for(i=0;i<=(model[fileloop])->metalorder;i++)
   for(j=0;j<(model[fileloop])->ligandorder+1;j++)
        *(beta[i]+j)=1.0;

  *(beta[0]+0)=0.0;   /*nothing present for complexation */

  metalsize=(model[fileloop])->metalorder;
```

```
 ligsize=(model[fileloop])->ligandorder;
 tailend=metalsize*ligsize;
 metind=1;
 ligind=1;

for(i=0;i<tailend;i++)
 {
  if(ligind>(model[fileloop])->ligandorder)
   {
    ligind=1;
    metind++;
   }
  if(i==parmnum)
    temphold=parmvalue;
  else
    temphold=*((model[fileloop])->string+i);

  *(beta[metind]+ligind)=temphold;
  ligind++;
}

/* omit the section for self-complexation of the ligand and/or metal */

i=filepoint;

  metalconc=*(totmet+i);
  metal= equilcalc(beta,i);

  sum=metalconc/metal;
  calce=log(sum)*SCONV;

  return(calce);

}
/*****************************************************/
/*********** Start of equil.c program merger **************/
double equilcalc(double **beta,int dataind)
{
 static int alreadydone=0;

 int i,j,k,kind,l,metind,ligind;
 int maxm,maxn,maxv,vtemp;
 static double *vv, *vp, *x, *ki, *logk  ;
 static double *c0, *c ;
 double sum;

 for (i=0; i < NMAX ; i++)          /* initialise the v matrix to zero for */
  for (j=0; j < MMAX ; j++)         /* elements not involved in a particular */
   v[i][j]=0.0;                     /* equilibrium */
```

```
 nequil=(model[fileloop])->metalorder*(model[fileloop])->ligandorder;  /* number of
equilibria */
 mequil= nequil +2;  /* number of species    */

 maxm=MAXSPECIES;
 maxn=MAXEQUIL;
 maxv=maxm*maxn;
 metind=1;
 ligind=1;
 /* Filling The V Matrix */
  for(i=0;i<nequil;i++)
   {
   for(j=0;j<mequil;j++)
    {
       if(i>=(model[fileloop])->ligandorder)
        {
         if(ligind >(model[fileloop])->ligandorder)
          {
            ligind=1;
            metind++;
          }
        }

       if(j==0)
         v[i][j]=metind;
       if(j==1)
         v[i][j]=ligind;

       if(metind==1)
        {
         if(j==(v[i][0]*v[i][0] + v[i][1]))
           v[i][j]=-1;
        }
       else
        {
        if((model[fileloop])->ligandorder >=1)
          {
          vtemp=v[i][0] +v[i][1] + (model[fileloop])->ligandorder;
          if( j==(vtemp -1))
              v[i][j] = -1;
          }
        }

    }
   ligind++;
   }

 if(!alreadydone)
  {
  vv=malloc(maxv*sizeof(double));
```

```
 if (vv==NULL) fault(401);
 vp=malloc(maxv*sizeof(double));
if (vp==NULL) fault(402);
x=malloc(maxn*sizeof(double));
if (x==NULL) fault(403);
ki=malloc(maxn*sizeof(double));
if (ki==NULL) fault(404);
logk=malloc(maxn*sizeof(double));
if (logk==NULL) fault(405);
c0=malloc(maxm*sizeof(double));
if (c0==NULL) fault(406);
c=malloc(maxm*sizeof(double));
if (c==NULL) fault(407);
alreadydone=1;
}

for(i=0; i< mequil; i++)
   c0[i]=0.0;
m0=2;

c0[0]=*(totmet+dataind);
c0[1]=*(ligconc+dataind);

/*  printf("\n\nEquil. constants\n\n");    */
 kind=0;
 for(i=1;i<=(model[fileloop])->metalorder;i++)
   for(j=1;j<=(model[fileloop])->ligandorder;j++)
    {
     ki[kind]=*(*(beta+i)+j);
     kind++;
    }
  /* convert from log in the base 10 to log in the base e */

  for(i=0;i<kind;i++)
     logk[i]= ki[i]*log(10.0);
/*************************************************/

/* here fill vv matrix
*/

for (i=1; i <= nequil ; i++)
  for (j=1; j <= mequil; j++)
    *elem(vv,mequil,i,j) = *elem(v[0],MMAX,i,j);

guess(nequil,mequil,vv,c0,x,c);

for(i=0;i<nequil*mequil; i++)
      vp[i]=vv[i];
iter(nequil,mequil, vp,c,x,logk);
  printf("\n");
```

```
/* print results */

/*printf("\nResults\n*******\n\n");
printf("Species                CO              C\n");
printf("-------------------------------------------\n");
for (i=0;i<mequil; i++)
 printf("component %d %15.4e %15.4e\n",i,c0[i],c[i]);  */

return(c[0]);


}

/****** end of main calling routine for the EQUIL.C program ************/

/* DIAG() ROUTINE : returns the value of diagonal element of a square */
/* matrix of doubles of size nequil numbering of elements starts at 1 */

double *diag( double *a, int nequil, int diagnr)
{
  double *diagonal ;
  diagonal= a+ ((diagnr-1)*(nequil+1));
  return (diagonal);
}
/*********************************************************************/

/* ELEM() ROUTINE :returns the value of the matrix element of a matrix with */
/* row size nequil and with indices row, column numbering starts at 1      */

double *elem( double *a, int nequil, int row, int col)
{
  double *element ;
  element = a + (nequil*(row-1) + col-1) ;
  return(element);
}
/*********************************************************************/

/************* MINA() returns the minimum of two values *******************/
double mina(double a, double b)
 {
 return( (a >= b) ? b : a);
 }

/*********************************************************************/

/************* MAXA() returns the maximum of two values ******************/
double maxa(double a, double b)
 {
  return( (a >= b) ? a : b);
```

```
 }
/***********************************************************************/

/************** FAULT() prints error message as stated in paper *************/
void fault(int nequil)
 {
  printf("ERROR= %d\n",nequil);
  exit(nequil);
 }
/***********************************************************************/

/*************** DETSYM() ROUTINE START ***************************/
void detsym( double *a, int nequil, double *minpiv)
{
  double r, s ;
  int k,l,j ;
  for ( k=1; k <= nequil ; k++)
   {
    r=  *diag(a, nequil, k);
    for (l=1; l <= k-1 ; l++)
         r= r - *elem(a, nequil,l,k)*(*elem(a,nequil,l,k));
    if (k==1) *minpiv=r;
    *minpiv=mina(*minpiv,r);
    if ( r <= 0.0) return;
    *diag(a,nequil,k)=r=sqrt(r);
    for (j=k+1; j <= nequil ; j++)
        {
         s=*elem(a,nequil,k,j) ;
         for (l=1; l <= k-1; l++)
           s=s-(*elem(a,nequil,l,j))*(*elem(a,nequil,l,k));
         *elem(a,nequil,k,j)= s/r;
        }
    }
}
/***********************************************************************/

/************* SOLSYM() FUNCTION START ***************************/
void solsym(double *a, int nequil, double *b)
{
  double s;
  int i,k ;
  for (i=1; i <= nequil; i++)
  {
    s= b[i-1];
    for (k=1; k <= i-1; k++)
         s=s - *elem(a,nequil, k,i)*b[k-1];
    b[i-1]=s/(*diag(a,nequil,i));
  }
  for (i=nequil; i >= 1; i--)
  {
```

```
        s=b[i-1];
        for (k=i+1; k <=nequil ; k++)
          s=s- (*elem(a,nequil,i,k)*b[k-1]);
        b[i-1]=s/(*diag(a,nequil,i));
  }
}
/**********************************************************************/

/***** START OF INITIAL GUESS FOR EQUILIBRIUM CONCENTRATION ****/
void guess(int nequil, int mequil, double *v, double *c0 , double *x, double *c)
{
  double c0sum, at, bt, an, bn, a, b ;
  int i, j, first, ready ;
  for (i=0; i < nequil ; i++)
     x[i]=0.0;
  c0sum=0.0 ;
  for (i=0; i < mequil; i++)
  {
   c0sum += c0[i];
   c[i]=c0[i];
  }
LB1: first=0;
   ready=1;

 for(i=1; i <=nequil ; i++)
  {
  if(x[i-1] != 0.0 ) goto LB2;
  at=bt=c0sum;
  an=bn=0.0;
  for(j=1;j<=mequil;j++)
   {
     if(*elem(v,mequil,i,j) > 0.0 )
        {
         at=mina(at,c[j-1]);
         an=maxa(an,*elem(v,mequil,i,j));
         }
     if(*elem(v,mequil,i,j) < 0.0 )
         {
          bt=mina(bt,c[j-1]);
          bn=maxa(bn,-(*elem(v,mequil,i,j)));
         }
   }

 if((an==0.0) || (bn==0.0)) fault(21) ;
 a=at/an;
 b=bt/bn;
 if((a==0.0) && (b==0.0))
  {
   ready=0;
   goto LB2;
```

```
   }

  if((a !=0.0) && (b != 0.0)) goto LB2;

  first=1;
  x[i-1]=0.5*(a-b);
  for(j=1;j<=mequil;j++)
   {
    c[j-1]=c[j-1]-x[i-1]*(*elem(v,mequil,i,j));
   }
  LB2:  continue ;
 }
if (ready==1) return;
if (first==1) goto LB1;
fault(22);
return;
}
/*****************************************************************/

/* ITER() ROUTINE: to iteratively improve the estimate of the equilibrium   */
/* concentration of the species described in the V matrix */

void iter(int nequil, int mequil, double *v, double *c, double *x, double *logk)
{
   double tolc, epsc, p0, minpiv, fmax0, fmax1, wr, alfa ;
   int i,j, k, it, itmax ;
   static double *w, *z, *f, *dx, *dc, *logc, sum;
   static int alreadydone=0;
   int maxm=10;
   int maxv=50;
   int maxn=5;
 /* here allocations for dynamic array storage */

   if(!alreadydone)
     {
        w= malloc(maxv*sizeof(double));
        if (w==NULL) fault(201);
        z=malloc(maxv*sizeof(double));
        if (z==NULL) fault(202);
        f=malloc(maxn*sizeof(double));
        if (f==NULL) fault(203);
        dx=malloc(maxn*sizeof(double));
        if (dx==NULL) fault(204);
        dc=malloc(maxm*sizeof(double));
        if (dc==NULL) fault(205);
        logc=malloc(maxm*sizeof(double));
        if (logc==NULL) fault(206);
        alreadydone=1;
     }
   itmax=1000;
```

```
tolc=1.0e-10;
p0=1.0e-3;
for (i=1; i <= nequil; i++)
 for (j=1; j <= i ; j++)
       {
        sum = 0.0 ;
         for (k=1; k <= mequil ; k++)
                sum += (*elem(v,mequil,i,k))* (*elem(v,mequil,j,k));
        *elem(z,nequil,i,j)= *elem(z,nequil,j,i)= sum;
       }
detsym(z,nequil, &minpiv);
wr= *elem(z,nequil,1,1);
for (i=2;i <=nequil ; i++)
   wr=wr* (*diag(z,nequil,i));
if ((minpiv <= 0.0) || (wr < 0.5)) fault(31);
fmax0=0.0;
for (j=0; j< mequil ; j++)
  {
/*  printf(" c[%d] is %e \n",j,c[j]);  */
  logc[j]=log(c[j]);
  }
 for (i=1; i <= nequil ; i++)
  {
      sum=0.0;
      for (k=1; k<= mequil; k++)
       sum += *elem(v,mequil,i,k)* logc[k-1];
      f[i-1]=logk[i-1]+sum;
      fmax0=maxa(fmax0, fabs(f[i-1]));
  }
for (it=1 ; it<= itmax ; it++)
{
  printf(".");
  for (j=1; j <= mequil ; j++)
   for (k=1; k<= nequil; k++)
      *elem(w,mequil,k,j)= *elem(v,mequil,k,j)/ c[j-1];

   for (i=1; i <= nequil ; i++)
   for (j=1; j<= i; j++)
   {
      sum=0.0;
      for (k=1; k<= mequil; k++)
       sum += *elem(w,mequil,i,k)*(*elem(v,mequil,j,k));
       *elem(z,nequil,i,j) = *elem(z,nequil,j,i) = sum;
   }
  detsym(z,nequil,&minpiv);
  if (minpiv <= 0.0) fault(32);
  solsym(z,nequil,f);
  for (i=1; i <= nequil; i++)
        {
          dx[i-1]=f[i-1];
```

```
        }
    alfa=1.0;
    epsc=0.0;
    for (j=1; j <=mequil ; j++)
    {
        sum=0.0;
        for (k=1; k<= nequil; k++)
          sum += *elem(v,mequil,k,j)* dx[k-1];
        dc[j-1]=-sum;
/*      printf("dc[%d] is % e\n",j,dc[j-1]); */
        wr=dc[j-1]/c[j-1];
/*      printf("wr is %e epsc is %e \n",wr, epsc); */
        epsc=maxa(epsc,fabs(wr));
        if ( (1.0+ alfa*wr) < p0) alfa=(p0-1.0)/wr;
    }
    if (epsc < tolc )
        {
          return;
        }
LC:  for (j=1; j<=mequil; j++)
        {
          logc[j-1]=log(c[j-1]+alfa*dc[j-1]);
/*        printf("logc[%d] is %e ,alfa is %e \n",j,logc[j-1],alfa); */
        }
    fmax1=0.0;
    for (i=1; i <=nequil ; i++)
    {
        sum=0.0;
        for (k=1;k <=mequil ; k++)
          {
           sum += *elem(v,mequil,i,k)*logc[k-1];
          }
        f[i-1]=logk[i-1]+sum;
        fmax1=maxa(fmax1,fabs(f[i-1]));
    }
    if ( fmax1 > fmax0 )
        {
          alfa *= 0.5;
          goto LC ;
        }
    fmax0=fmax1;
    for (k=1; k<=nequil; k++)
        x[k-1] += alfa*dx[k-1];
    for (k=1; k <=mequil ; k++)
        c[k-1] += alfa*dc[k-1];
    }
    for (k=1; k<=nequil ; k++)
        printf("%g ",x[k-1]);
    printf("\n");
    fault(33);
```

```
	return;
}

/****************************************************************************/

/* This is file INDEX.C */
/* This file may have been modified by DJ Delorie (Jan 1991).  If so,
** these modifications are Coyright (C) 1991 DJ Delorie, 24 Kirsten Ave,
** Rochester NH, 03867-2954, USA.
*/

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted provided
 * that: (1) source distributions retain this entire copyright notice and
 * comment, and (2) distributions including binaries display the following
 * acknowledgement:  ``This product includes software developed by the
 * University of California, Berkeley and its contributors'' in the
 * documentation or other materials provided with the distribution and in
 * all advertising materials mentioning features or use of this software.
 * Neither the name of the University nor the names of its contributors may
 * be used to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#if defined(LIBC_SCCS) && !defined(lint)
static char sccsid[] = "@(#)index.c	5.6 (Berkeley) 6/23/90";
#endif /* LIBC_SCCS and not lint */

#include <string.h>
#include <stddef.h>

char *
#ifdef STRCHR
strchr(p, ch)
#else
index(p, ch)
#endif
	register const char *p, ch;
{
	for (;; ++p) {
		if (*p == ch)
			return((char *)p);
```

```
            if (!*p)
                    return((char *)NULL);
        }
        /* NOTREACHED */
}
/**********************************************************************/
/**********************************************************************/
/* Given a matrix A[1-m][1-n], calculate its singular value decomposition */
/* A=U W V(t), matrix U replaces A on output from this function. The     */
/* diagonal matrixof singular values W is output as a vector w[1-n]      */
/**********************************************************************/

void svdcmp(double **a,int m,int n,double *w,double **svd)
{

 int flag,i,its,j,jj,k,l,nm;
 double anorm,c,f,g,h,s,scale,x,y,z,*rv;
 double temp;

if((rv=(double *)malloc(n*sizeof(double)))==NULL)
 {
   printf("not enough memory to allocate to rv vector \n");
   exit(1);
  }

g=scale=anorm=0.0;
 for(i=1;i<=n;i++)
  {
  l=i+1;
  rv[i]=scale*g;
  g=s=scale=0.0;
  if(i<=m)
   {
   for(k=i;k<=m;k++)
        scale +=fabs(a[k][i]);
   if(scale)
    {
       for(k=i;k<=m;k++)
         {
          a[k][i]/=scale;
          s +=a[k][i] *a[k][i];
         }
       f=a[i][i];
       g= -SIGN(sqrt(s),f);
       h= f*g -s;
       a[i][i] = f-g;
       for(j=l;j<=n;j++)
        {
         for(s=0.0,k=i;k<=m;k++)
           s +=a[k][i] *a[k][j];
```

```
          f=s/h;
          for(k=i;k<=m;k++)
            a[k][j] +=f*a[k][i];
          }
      for(k=i;k<=m;k++)
          a[k][i] *=scale;

      } /*end of if scale */
    }/* end of if i<=m */
   w[i]=scale *g;
 g=s=scale=0.0;

 if(i<=m && i !=n)
   {
    for(k=l;k<=n;k++)
         scale +=fabs(a[i][k]);
     if(scale)
          {
          for(k=l;k<=n;k++)
            {
             a[i][k] /=scale;
             s +=a[i][k] *a[i][k];
            }
           f=a[i][l];
           g= -SIGN(sqrt(s),f);
           h=f*g-s;
           a[i][l]=f-g;
           for(k=l;k<=n;k++)
             rv[k]=a[i][k]/h;
           for(j=l;j<=m;j++)
             {
              for(s=0.0,k=l;k<=n;k++)
                s+=a[j][k]*a[i][k];
              for(k=l;k<=n;k++)
                a[j][k] +=s*rv[k];
             }
         for(k=l;k<=n;k++)
           a[i][k] *=scale;

          } /* end of if scale loop */
   } /* end of if i<=m */

   anorm=FMAX(anorm,(fabs(w[i])+fabs(rv[i])));
  }  /* end of for i=  loop */

/* accumulation of right hand transformations */
for(i=n;i>=1;i--)
 {
  if(i<n)
   {
```

```
    if(g)
     {
        for(j=l;j<=n;j++)
         svd[j][i]=(a[i][j]/a[i][l])/g;
        for(j=l;j<=n;j++)
          {
           for(s=0.0,k=l;k<=n;k++)
             s+= a[i][k] *svd[k][j];
           for(k=l;k<=n;k++)
             svd[k][j] +=s*svd[k][i];
          }
     } /* end of if g */
   for(j=l;j<=n;j++)
     svd[i][j]=svd[j][i]=0.0;
  } /* end of if i < n */
 svd[i][i]=1.0;
 g=rv[i];
 l=i;
}/* end of for i=1->n */

for(i=IMIN(m,n);i >=1;i--)
 {
  l=i+1;
  g=w[i];
  for(j=l;j<=n;j++)
    a[i][j]=0.0;
  if(g)
   {
    g= 1.0/g;
    for(j=l;j<=n;j++)
         {
          for(s=0.0,k=l;k<=m;k++)
            s +=a[k][i]*a[k][j];
          f=(s/a[i][i])*g;
          for(k=i;k<=m;k++)
            a[k][j] +=f*a[k][i];
     }

    for(j=i;j<=m;j++)
         a[j][i] *=g;
   }
  else
   for(j=i;j<=m;j++)
         a[j][i]=0.0;

  ++a[i][i];
 }

for(k=n;k>=1;k--)
 {
```

```
for(its=1;its<=30;its++)
 {
  flag=1;
  for(l=k;l>=1;l--)
       {
        nm=l-1;
        if((fabs(rv[l])+anorm)==anorm)
          {
           flag=0;
           break;
          }
        if((fabs(w[nm])+anorm) ==anorm)
          break;
       }
 if(flag)
      {
       c=0.0;
       s=1.0;
       for(i=l;i<=k;i++)
        {
         f=s*rv[i];
         rv[i]=c*rv[i];
         if((fabs(f)+anorm)==anorm)
           break;
         g=w[i];
         h=pythag(f,g);
         w[i]=h;
         h=1.0/h;
         c=g*h;
         s=-f*h;
         for(j=1;j<=m;j++)
           {
            y=a[j][nm];
            z=a[j][i];
            a[j][nm]=y*c +z*s;
            a[j][i]=z*c -y*s;
           }
        } /* end of for i=1->k */
      }/* end of if flag */

 z=w[k];
 if(l==k)
  {
     if(z<0.0)
      {
       w[k] = -z;
       for(j=1;j<=n;j++)
         svd[j][k]= -svd[j][k];
      }
     break;
```

```
 }
if(its==30)
    {
     printf("no convergence after 30 iterations \n");
     exit(1);
    }

x=w[l];
nm=k-1;
y=w[nm];
g=rv[nm];
h=rv[k];
f=((y-z)*(y+z)+(g+h)*(g-h))/(2.0*h*y);
g=pythag(f,1.0);
f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
c=s=1.0;
for(j=l;j<=nm;j++)
 {
    i=j+1;
    g=rv[i];
    y=w[i];
    h=s*g;
    g=c*g;
    z=pythag(f,h);
    rv[j]=z;
    c=f/z;
    s=h/z;
    f=x*c +g*s;
    g= g*c -x*s;
    h=y*s;
    y*=c;
    for(jj=1;jj<=n;jj++)
     {
      x=svd[jj][j];
      z=svd[jj][i];
      svd[jj][j]=x*c +z*s;
      svd[jj][i]= z*c -x*s;
     }
    z=pythag(f,h);
    w[j]=z;
    if(z)
     {
      z=1.0/z;
      c=f*z;
      s=h*z;
     }
    f=c*g +s*y;
    x=c*y -s*g;
    for(jj=1;jj<=m;jj++)
     {
```

```
            y=a[jj][j];
            z=a[jj][i];
            a[jj][j]=y*c +z*s;
            a[jj][i]=z*c -y*s;
           }
       }/* end of for j=1->m */
       rv[l]=0.0;
       rv[k]=f;
       w[k]=x;

   }
  }
}
/**********************************************************************/
/* Calculates sqrt(a*a +b*b) without underflow or overflow problems      */
/**********************************************************************/

double pythag(double a,double b)
{
 double absa,absb,output,outputtemp;
 absa=fabs(a);
 absb=fabs(b);
 if(absa > absb)
  {
   outputtemp=absb/absa;
   output=absa*sqrt(1.0+outputtemp*outputtemp);
   return(output);
  }
 else
  {
   if(absb==0.0)
     return(0.0);
   else
    {
       outputtemp=absa/absb;
       output=absb*sqrt(1.0+outputtemp*outputtemp);
       return(output);
    }
  } /* end of outer else */

 } /* end of function */

/**********************************************************************/

int IMIN(int a,int b)
{
 int iminarg1,iminarg2;
 iminarg1=a;
 iminarg2=b;
 if(iminarg1 >=iminarg2)
```

```
   iminarg1=iminarg2;
 return(iminarg1);
 }
/**********************************************************************/
double SIGN(double a, double b)
 {
  if(b>0.0)
    b=fabs(a);
  else
    b=-fabs(a);
  return(b);
}
```

□

Appendix 10 Formal Framework For Genetic Algorithms [1-8]

The formal framework for a genetic algorithm (GA) which mimics biological adaptation, defines the domain or the search space of the adaptive plan as $\alpha=\{A_1,A_2,....\}$ where A_i represents the individual chromosomes. Each chromosome A_i is composed of **d** genes $A_i=\{a_{i,1}, a_{i,2},.........a_{i,d}\}$ and the genes are expressed in the form of a uniform alphabet **V**. The number of characters in an alphabet is constant and is known as the cardinality of the alphabet **k**,(e.g. a binary alphabet has a cardinality of two). The phenotype of the chromosome is determined by the decryption of the problem parameters from the genotype. The action of the environment **E** on a particular phenotype (and hence genotype A α) is described by a performance measure known as the fitness μ_{E},,the number of offspring of a particular genotype in the next generation is determined by μ_E. Γ is the set of feasible or possible adaptive plans τ. The adaptive plan τ is taken to act at discrete instants of time **t=1,2,3,......** rather than continuously, and produces a sequence of structures or a trajectory through α, by making successive selections from a set of operators $\underline{\Omega}$, which are applied to probablistically chosen individuals **A** from the current population $\beta(t)$. The set of genetic operators $\Omega=\{w_1,w_2,.......\}$ have two major properties in common, the first of which is that the operators do not directly affect the size of the population, instead they mainly alter and redistribute alleles within the population. The second property of the operators is that they infrequently separate alleles which are close together on a chromosome. The operators which will be investigated in this study are reproduction, crossover and mutation.

10.1. Genetic Algorithm Operators

10.1.1. Schema and Reproduction

Optimisation by GA involves searching for similarities among strings in the population and determining causal relationships between the similarities of highly fit individuals and their fitness performance. Holland developed the idea of schemata to designate subsets of the population that have common attributes at certain string positions. Each schema is a string of the same length as the chromosomes in the population. The string is defined over an extended alphabet consisting of the alphabet of the original population and an additional metasymbol * whose presence at a particular position in a string indicates that any element from the original alphabet could exist at that position . As such if a population consists of strings of length L defined over an alphabet of cardinality k, then a schema is described as a string of length L defined over the extended alphabet of cardinality **k+1**. In order to exemplify this concept, a schema $\mathbf{v_1}$** indicates the subset of all the elements in the search space α which have the attribute $\mathbf{v_1 \in V}$ at position 1. Any string $\mathbf{A \in \alpha}$ belongs to the subset designated by the schema $\xi = (\Delta_{i,1}, \Delta_{i,2}, \ldots\ldots\ldots \Delta_{i,L}) \in \Xi$ if and only if two conditions are satisfied. The first condition stipulates that whenever $\Delta_{i,j} \in \xi = *$, any attribute from the alphabet **V** of the original population can exist at the jth position of **A**. The second condition requires that whenever $\Delta_{i,j} \in \mathbf{V}$, the same attribute $\Delta_{i,j}$ must occur at the jth position of **A**.

In general for alphabets of cardinality **k**, there are $\mathbf{(k+1)^L}$ schemata present in a search space defined by strings of length **L**. All schemata however are not the same. Some schema are more specific than others, the order of a schema $\mathbf{O(\xi)}$ indicates the

number of $\Delta_{i,j}$ present in the schema which have a fixed value derived from the original alphabet **V**. Some schema span more of the total string length than others. If the fixed positions on the schema ξ are given by $i_1<i_2<.......<i_h$ then the defining length of the schema $l(\xi)$ is given by $l(\xi)=(i_h-i_1)$.

If the search space of a particular problem, is expressed in the form of binary three element strings then the possible search points of this problem can be represented as the corners of a cube of dimension three. The schemata associated with this problem can also be represented as various geometric entities related to the hypercube.

Schemata of order 3 correspond to the actual points of the search space and hence the corner points (zero dimensional hyperplanes) of the cube. The schemata of order two (two specified positions and one *) correspond to the edges (one dimensional hyperplanes) of the cube and each schemata contains two points from the search space. Schemata with specificity 1 contain four points from the search space and correspond to the faces of the cube. The single schemata of specificity 0 consists of the cube itself.

Figures 10.1 and 10.2 graphically depict these geometric relations and table (10.1) describes the relationships between the order of a schema and its number in this problem case.

Schema Specificity $O(\xi)$	Hyperplane Dimension $(L-O(\xi))$	Geometric Realisation	Individuals In The Schema	Number Of Such Schemata
3	0	Point	1	8
2	1	Line	2	12
1	2	Plane	4	6
0	3	Entire Cube	8	1
			Total	27

Table 10.1 - Schemata in a binary 3 dimensional search space

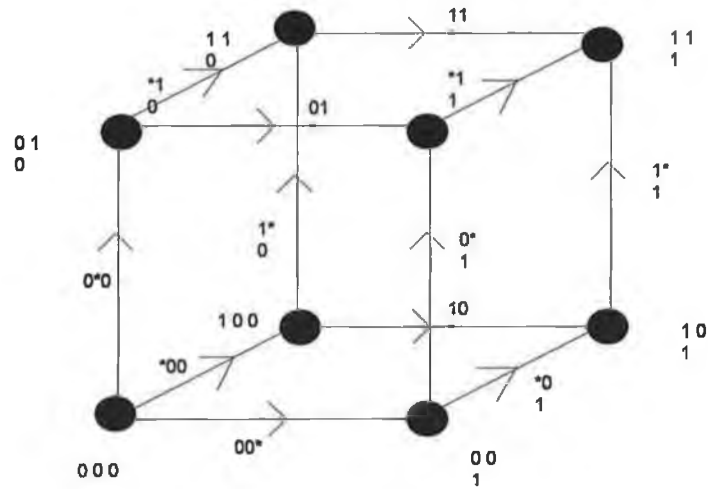


Figure (10.1) Schemata of specificity 3 and 2 in a 3 dimensional binary search space

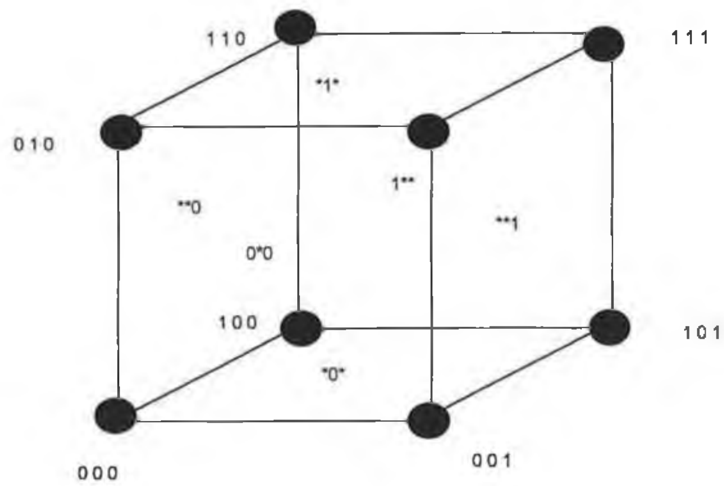


Figure (10.2) Schemata of specificity 1 in a 3 dimensional binary search space

Each individual **A** in the population belongs to 2^L schemata since either $\mathbf{a_i \in V}$ **or** $\mathbf{a_i} =*$ $\forall$ **i** an example of such set of schemata is depicted in figure (10.3).

0	*	*
*	1	*
*	*	0
0	1	*
0	*	0
*	1	0
0	1	0
*	*	*

Figure 10.3 - The eight schemata for the string 010

However usually more individuals in the population belong to schema of lower specificity. The GA implicitly allocates credit for the observed fitness of each explicitly tested individual in the population to all of the $\mathbf{2^L}$ schemata to which the individual belongs. A particular schema will usually receive allocations that contribute to its average fitness from a number of different individuals in the population. The **M** individual chromosomes in the population make a total of $\mathbf{M.2^L}$ contributions to the calculations of the $\mathbf{(k+1)^L}$ values of schemata fitness.

The object of the GA is to continue the search for an optimum in regions of the search space that consisted of points that belonged to highly fit schema. But the continued search should not merely test points that had already been explicitly tested, rather it should test new points from the search space that belong to the same highly fit schema.

A schema ξ is considered to be potentially useful for directing further progress through the search space if it's observed average performance $\tilde{\mu}_\xi$ is greater than the overall average performance of the schemata represented by the current population. However the observed average performance of a schema $\tilde{\mu}_\xi$ is basically a sample average of a random variable with an associated variance $\sigma(\varepsilon)$. As such there is always a non-zero probability for two schemata ξ and ξ' that $\mu_\xi > \mu_\xi'$ even though $\tilde{\mu}_\xi' < \tilde{\mu}_\xi'$. Confidence that $\mu_\xi > \mu_{\xi'}$ reflects $\tilde{\mu}_\xi > \tilde{\mu}_\xi$ can only be increased by allocating further trials to both ξ and ξ'. Thus further operation of the search can be employed to reduce the probability of error (i.e. the possibility that $\mu_{\xi'} > \mu_\xi$ even though $\tilde{\mu}_\xi' < \tilde{\mu}_\xi$) or to exploit the best observed schema for further exploration of the search space. This introduces the conflict between exploiting what is already known about the search space and exploring the search space in order to obtain further information.

The Two Armed Bandit problem (TAB problem) is an example of a similar decision making dilemma. This model involves a two armed slot machine in which one arm pays an award of μ_1 with variance σ_1^2 and the other arm pays an award μ_2 with variance σ_2^2. The goal of this problem is to maximise the payoff of the slot machine while playing over a period of time. If one knew for certain that one arm provided a better payoff than the other, the optimal strategy would be trivial, one would play the arm with the highest payoff 100 % of the time. Without this knowledge, a certain number of trials could be allocated to each arm in order to learn something of their relative payoffs.

With a total of N trials to allocate among the two arms, an equal number of trials **n** **($2n < N$)** could be allocated to each of the two arms during the experimental

phase. After the experiment, the remaining **N-2n** trials could then be allocated to the arm with the best observed performance. There are two different sources of loss associated with this strategy. The first source of loss arises if the observed best arm ξ_1 is actually second best, in which case the **(N-n)** trials allocated to ξ_1 would incur an expected cumulative loss of $(N-n)|\mu_1-\mu_2|$. The probability of this happening is given by **q(N-n, n)**. The second source of loss arises when the observed best ξ_1 is in fact the best arm, in which case the n trials allocated to ξ_2 incurs a loss of $n|\mu_1-\mu_2|$. Hence the total expected loss for any allocation of n trials to ξ_2 and **N-n** trials to ξ_1 is given by

$$L(N-n,n)=[q(N-n,n).(N-n)+(1-q[N-n,n]).n].|\mu_1-\mu_2| \qquad (10.1)$$

The probability **q(N-n,n)** is approximated by the tail of the normal distribution

$$q(N-n,n)=\frac{1}{\sqrt{2\pi}}\frac{e^{-x^2/2}}{x} \qquad (10.2)$$

where

$$x=\frac{\mu_1-\mu_2}{\sqrt{\sigma_1^2+\sigma_2^2}}\sqrt{n} \qquad (10.3)$$

Holland showed that given **N** trials to be allocated to two random variables ξ_1 and ξ_2 with means μ_1 and μ_2 and variances $\sigma_1{}^2$ and $\sigma_2{}^2$ respectively, the minimum expected loss results when the number of trials allocated to ξ_2 is given by

$$n\le n^*\approx b^2\ln\left[\frac{N^2}{8\pi b^4\ln N^2}\right] \qquad (10.4)$$

The expression can be rearranged to yield the number of trials allocated to ξ_1 by

$$N-n^*\equiv N\equiv\sqrt{8\pi b^4\ln N^2}\,.e^{n^*/(2b^2)} \qquad (10.5)$$

Thus the loss rate will be optimally reduced if the number of trials allocated to ξ_1 grows slightly faster than an exponential function of the number of trials allocated to ξ_2. This is true irrespective of the form of the distributions defining ξ_1 and ξ_2.

However solving a simple TAB problem is no longer sufficient when dealing with GAs since the usual GA involves the simultaneous solution of multi-armed bandit problems. Hence a general performance criterion for adaptive plans must treat loss rates for an arbitrary number of options and involves an extension of the TAB problem to a k-armed bandit problem.

Under the same conditions as for the TAB problem, but now with **k** random variables, the minimum expected loss after **N** trials must exceed

$$(\mu_1 - \mu_2).(k-1)b^2\left[2+\ln\left(\frac{N^2}{8\pi(k-1)^2 \ln N^2}\right)\right] \qquad \textbf{(10.6)}$$

The minimal expected loss solution to the allocation of trials to k competing arms is similar to the armed TAB solution as it dictates that greater than exponentially increasing numbers of trials be given to the observed best of the **k** arms.

Schemata can be viewed as being in competition with each other (like the possible pulling strategies of a k-armed bandit problem). Two schemata A and B with individual positions a_i and b_i are competing if at all positions **i=1,2,....l** either $\mathbf{a_i=b_i=*}$ or $\mathbf{a_i \neq b_i \neq *}$ (i.e. $\mathbf{a_i \neq b_i}$ for at least one i value). In terms of competing schemata, the optimal way to allocate trials (population members in the next generation) is to allocate an approximately exponentially increasing number of future trials to a schema on the basis of the ratio of the current estimate of the average fitness of the schema to the current estimate of the population average fitness.

Since there are $\binom{L}{j}$ ways of choosing j positions along a string of length L

there are $\binom{L}{j}$ different k-armed bandit problems where $\mathbf{k_j = 2^j}$. The optimal allocation

of trials should occur simultaneously to all $\mathbf{(k+1)^L}$ possible schemata from the current

population. Holland's theorem of GA or schema theorem demonstrates that the GA

satisfies all these $\mathbf{(k+1)^L}$ constraints.

Darwinian fitness proportional reproduction indicates that a string is copied

into the next generation according to its fitness as follows

$$\mathbf{P_i = \frac{\mu_i}{\sum_{i=1}^{M} \mu_i}} \qquad \textbf{(10.7)}$$

$\mathbf{P_i}$= probability of string i being selected

μ_i= fitness of string i

If there are $\mathbf{M_\xi(t)}$ examples of schema ξ in the population at time t, the Darwinian

reproduction strategy affects the schema such that the number of representatives of the

schema in the population at time **t+1** is given by

$$\mathbf{M_\xi(t+1) = M_\xi(t).M.\frac{\tilde{\mu}_\xi(t)}{\sum \mu_i(t)}} \qquad \textbf{(10.8)}$$

where

$$\mathbf{\tilde{\mu}_\xi(t) = \frac{\sum_{x_i \in \xi} \mu(x_i, t)}{M_\xi(t)}} \qquad \textbf{(10.9)}$$

is the average performance of the strings representing ξ at time **t**.

But since the average fitness of the entire population can be rewritten as

$$\mathbf{M_\xi(t+1) = M_\xi(t).\frac{\tilde{\mu}_\xi(t)}{\tilde{\mu}(t)}} \qquad \textbf{(10.10)}$$

Hence the use of Darwinian reproduction causes a particular schema to grow as the ratio of the average fitness of the schema to the average fitness population.

If the average fitness of a schema ξ remains above the average fitness of the population $\tilde{\mu}(t)$ by an amount $c\tilde{\mu}(t)$ [c is a constant] then the schema difference equation can be rewritten as

$$M_\xi(t+1) = M_\xi(t).\frac{\tilde{\mu}(t) + c\tilde{\mu}(t)}{\tilde{\mu}(t)} = (1+c)M_\xi(t) \qquad (10.11)$$

starting at time **t=0** produces the expression

$$M_\xi(t) = M_\xi(0).(1+c)^t \qquad (10.12)$$

which indicates that reproduction allocates exponentially increasing numbers of copies in future generations to above average schema (which is close to the optimal allocation of trails predicted by the k-armed bandit problem).

10.1.2 Crossing Over

In biological systems, crossover is a process which causes recombination of alleles via exchange of segments of genetic code between pairs of chromosomes. The GA scheme for mimicking this process progresses in three steps. In the first step two strings $A=v_1v_2.......v_L$ and $A'=v'_1v'_2.....v'_L$ are selected (usually at random) from the current population $\beta(t)$. The second step involves the selection of a number x at random from the set **{1,2....,L-1}**, this number corresponds to the position on the two strings **a** and **a'** at which crossover will occur. In the third step, two new structures are formed from **A** and **A'** by exchanging the set of attributes to the right of position x, the resulting structures have the form $v_1...v_xv_{x+1}..v_L'$ and $v_1'...v_x'v_{x+1}...v_L$.

The effects of crossing over on the current population include the generation of new instances of schemata already existing in the population and the generation of new schemata. If $A=v_1v_2...v_L$ and $A'=v_1'v_2'.......v_L'$ differ in attribute values at x' positions to the left of the crossing position x+1 and x" positions to the right of x, the result of a single crossing over of A with A' at x will be $2^L - 2^{L-x'} - 2^{L-x''} + 2^{L-(x'+x'')}$ new schemata and $2^{L-x'} + 2^{L-x''} - 2^{L-(x'+x'')}$ schemata of which A and A' are already subsets.

Linkage arises because a schema is less likely to be affected by crossover if its defining positions are closer together on the chromosome string. The probability of a crossover occurring in between the region spanned by two defining positions is given by $\frac{L(\xi)}{L-1}$ where $L(\xi)$ is the defining length of the schema. Hence the smaller the defining length of the schema, the less likely it is to be affected by crossover. Crossing a schema ξ with an identical schema ξ_1 will not destroy either schema, however if $\xi \neq \xi_1$ the result of the cross may or may not disrupt both schema depending on where the crossover occurs.

Let the probability of crossing be **Pc**, then the probability of crossing inside the defining length of a schema is given by $\frac{Pc.L(\xi)}{L-1}$. The proportion of schema ξ in the current population $\beta(t)$ is given by $P(\xi, t) = \frac{M_\xi(t)}{M}$ and the proportion of the population which is not schema ξ is given by $1-P(\xi,t)$. Hence the probability of crossing causing the loss of the schema ξ is given by

$$\left(1 - P(\xi, t)\right).Pc.\frac{L(\xi)}{L-1} \qquad (10.13)$$

The number of the schema ξ which will be expected to appear in the next generation when combining fitness proportional reproduction with the disruptive effect of crossover can thus be determined as

$$\begin{aligned} P(\xi,t+1) &= \frac{M_\xi(t+1)}{M} \geq \left[1-(1-P(\xi,t)).Pc.\frac{L(\xi)}{L-1}\right].\frac{M_\xi'(t)}{M} \\ &= \left[1-(1-P(\xi,t)).Pc.\frac{L(\xi)}{L-1}\right].\frac{\mu_\xi(t)}{\tilde{\mu}(t)}P(\xi,t) \\ &= \left[1-Pc.\frac{L(\xi)}{L-1}(1-P(\xi,t))\right].\frac{\mu_\xi(t)}{\tilde{\mu}(t)}P(\xi,t) \end{aligned} \quad (10.14)$$

From this expression it can be seen that schema of short defining length have least effect on the near optimal allocation of schema copies in the next generation. Crossing over acts by introducing new schemata for testing, and by retesting existing schemata without disturbing the process of optimal allocation of trial in the next generation of a given schema to short defining length schema.

10.1.3 Mutation

In biological systems mutation is a process whereby one allele of a gene is randomly replaced by or modified to another allele to yield a new chromosome. Mutation in a GA operates on chromosome $A=v_1v_2.....v_L$ in the current population $\beta(t)$ by choosing positions $x_1,x_2,......x_h$ on the string at which mutation is to occur. These positions are chosen randomly where each position has a small probability of undergoing mutation independently of what happens at the other positions. A new structure

$\mathbf{A'} = \mathbf{v_1 .....v_{x_1 -1} v'_{x_1} v_{x_1 +1} .....v_{x_2 -1} v'_{x_2} v_{x_2 +1} ....v_L}$ is formed whereby $\mathbf{v'_{x_i}}$ is randomly drawn from the range of the alphabet **V** for each position $\mathbf{x_i}$.

If successive populations are produced solely by mutation (without fitness proportional selection or crossover) the result is a random sequence of strings drawn from the search space α. The process is enumerative since the order in which the strings are generated is unaffected by the observed performance of the strings. Since enumerative searching strategies have limited use, mutation's primary role is not one of generating new strings for testing. In a population β that is small relative to the search space α, there is always a possibility that the last copy of some allele will be lost from the population during the operation of the GA. Alleles which occur in structures of below average performance are particularly susceptible to loss, however at some later stage in the operation of the GA these alleles may be required in a different combination of other alleles for further improvement of the string's performance. Once an allele is lost from a population, the crossover operator has no way of reintroducing it. Hence mutation plays a role of ensuring that no allele is permanently lost to the population.

Mutation also acts as an additional source of loss of schemata undergoing reproduction. If the probability of mutation at each position $\leq \mathbf{Pm}$, then a schema defined on $\mathbf{O(\xi)}$ can expect to undergo one or more mutations with the probability

$$\mathbf{1-(1-Pm)^{O(\xi)}} \qquad \mathbf{(10.15)}$$

Thus if the mutation operator is added to the fitness proportional reproduction and crossover operators, the number of the schema ξ to appear in the next generation will be determined by

$$\mathbf{P}(\xi, t+1) \geq \left(\frac{1-\mathbf{Pc}.\mathbf{L}(\xi)}{\mathbf{L}-1}\right)[1-\mathbf{P}(\xi, t)]\,\frac{\tilde{\mu}_{\xi}(t)}{\tilde{\mu}(t)}.(1-\mathbf{Pm})^{o(\xi)}.\mathbf{P}(\xi, t) \qquad (10.16)$$

Hence short, low order, above average schemata receive exponentially increasing trials in subsequent generations, this is known as the Schema Theorem or Fundamental Theorem of Genetic Algorithms.

Genetic algorithms superficially seem to process only the strings present in the current population, but in actuality are implicitly processing in parallel a huge amount of information concerning unseen hyperplanes or schemata. Hence the genetic algorithm has the property of implicit or intrinsic parallelism which enables it to create individual strings in the new population in such a way that the hyperplanes representing similar individuals are all automatically represented in the population in the ratio of the average fitness of the schema to the overall average performance of the population. This is achieved without any explicit calculations or storage other than the simple genetic algorithm operations acting on the individual strings in the population. This leads to a focused exploration of the search space in which the search is directed towards regions of the search space that contain strings of above average performance. However the population remains widely distributed over the search space, reducing the susceptibility of the search to languish in local optima.

It can be seen that the power of the genetic algorithm is not derived from the testing of individual strings in the population. Rather, it is obtained from the efficient

exploitation of the immensity of information that the testing of strings provides with respect to the interactions among the components of these strings. Specific configurations of component values or alleles observed to contribute to good performance of a string are propagated through the strings in future generations in a highly parallel fashion.

It is evident that genetic algorithms make much more extensive use of the information provided by the evaluation of candidate solutions to a problem than most other heuristic search methods. The hill-climbing algorithm test several candidate solutions and retains the most useful one. In the process it discards a vast amount of information concerning the combinations of features present in the unsuccessful candidates. In genetic algorithms in contrast, combinations of features present in the unsuccessful strings may still be transferred to other more successful strings.

10.2 Symbol Convention

α: Search space

A: A particular string attainable by the adaptive plan $A \in \alpha$

a_i: Genes on chromosome $a_i \in A_i$

$\beta(t)$: The current population

E: The environment

Ξ: The set of all schemata defined over the search space

ξ: A schema, $\xi \in \Xi$

i: Index of positions on a string

k: Cardinality of alphabet used to encode genes

L: Length of string

$L(\xi)$: Defining length of schema ξ

L(N-n, n): Total expected loss in a two-armed bandit problem

M: Number of strings in the population

$M_\xi(t)$: Number of instances of schema ξ in the population $\beta(t)$

N: Number of trials in k-armed bandit problems

n: Number of trials allocated to least successful competitor in two-armed bandit problem

$O(\xi)$: Order of schema ξ

P_i: Probability of string i being selected for reproduction

$P(\xi,t)$: Proportion of schema ξ in the population $\beta(t)$

Pc: Probability of crossing

Pm: Probability of mutation

q(N-n, n): Probability that the arm with the best observed average payoff has the worst expected value payoff in a two-armed bandit problem

t: Time

V: Set of attributes of each position on a string, i.e. the alphabet of cardinality k

W_i: Genetic operator

***:** Metasymbol

μ_i: Fitness of string i

μ_ξ: **Expected payoff of ξ**

Γ: **Set of all adaptive plans**

τ: **A particular adaptive plan $\tau \in \Gamma$**

Ω: **Set of operators for adaptive plan τ**

σ_ξ: **Variance of observed performances of schema ξ**

Δ_{ij}: **Position j on schema ξ**

v_i: **Attribute at position i , $v_i \in V$**

Bibliography

1. J.H. Holland, **Adaptation in Natural and Artificial Systems**, MIT Press, Bradford Books Edition, USA, 1992
2. C.B. Lucasius, Towards Genetic Algorithm Methodology in Chemometrics, Ph. D. thesis, Katholieke Universiteit Nijmegen, Netherlands, 1993.
3. D.E. Goldberg, **Genetic Algorithms in Search, Optimization and Machine Learning,** Addison-Wesley, Reading, MA., 1989.
4. G.E. Liepins and M.R. Hilliard, **Annals of Operations Research** 21 (1989) 31-58.
5. G.E. Liepins and M.D. Vose, **J.Expt. Theor. Artif. Intell.** 2 (1990) 101-115.
6. J.R. Koza, **Genetic Programming, On the Programming of Computers by Means of Natural Selection**, MIT Press, USA, 1992
7. K. de Jong, **Machine Learning**, 3, 1988, 121-138
8. J.M. Fitzpatrick and J.J. Grefenstette, **Machine Learning**, 3, 1988, 101-120

Acknowledgements

Writing a series of acknowledgements for this thesis is hellish because there does not appear any adequate way of acknowledging the help and support of the many generous and considerate people I encountered during the last four years. When moving between countries and disciplines I have had the honour and pleasure of meeting a broad range of people whose diverse work practices and research interests provided the challenges which allowed myself and my work to evolve.

In Dublin City University I would especially like to thank my supervisor Dr. Dermot Diamond whose tolerance and courage allowed me to move into the intriguing world of neural networks and genetic algorithms. I would like to thank Dr. Robert Forster for providing guidance and encouragement with a distinctive sense of humour. I would also like to thank Dr. Charles Markham in the Physics Department who suffered my apparently endless questions concerning the C programming language with such patience and generosity. I would like to thank all the technicians in the Chemistry Department for all their practical advice and help. I would finally like to thank all the postgrads, especially those in the laboratory WG30 who endured discussions concerning some of the more esoteric sides of neural networks and genetic algorithms with such good humour. I would especially like to thank Miss Margaret Stanley, for her long-standing support and encouragement (good luck with your own !!).

In the United Kingdom, I would like to thank my supervisor Professor Philip Barker whose vision and dynamism lead me into the world of neural networks. I would like to thank Dr. Karim Manjii and Sue Giller for listening to some of my more offbeat ideas. I would also like to thank the other member of the Interactive Systems Research Group whose conversations gave me a very different outlook on the roles of computers in education.

In The Netherlands, I would like to thank my supervisor Dr. M. Bos whose encouragement and contributions enriched my views of the practical applications of genetic algorithms and their relationships to more conventional approaches. I would like to thank Professor W. van der Linden for your encouragement and allowing me the privilege of working with such a remarkable research group. I would also like to thank the members of the research group for their encouragement and advice. I would especially like to thank Dr. Albert Bos who dealt with my professional and personal crises with such generosity and maddening calm. Thank you for your courage and tact when telling me the things I didn't want to hear and for being there when I needed someone to talk to.

Finally I would like to acknowledge the support and encouragement I received from my family. I couldn't have done it without you.