Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
A New Kernelized Approach to the Cerebellar Model Articulation Controller

Carl Laufer
Department of Electrical and Electronic Engineering
University of Auckland


September 2014
Abstract

In this thesis a new approach to the cerebellar model articulation controller (CMAC) is presented for use in online motion control learning. This new approach incorporates various CMAC enhancements that are found in the academic literature. The new CMAC is referred to as the "IKRLS-CMAC" or the "Improved Kernel Recursive Least Squares CMAC". This is because one of the main improvements is the change of learning algorithm from least mean squares (LMS) to kernel recursive least squares (KRLS). The advantage is that the KRLS algorithm learns faster.

The IKRLS-CMAC also implements several other enhancements. It is able to use higher order sensitivity functions, which overcome the CMACs discontinuous output problem. It uses credit assignment, also known as regularization to fix the CMACs well known learning interference problem, which can in some cases cause very poor modeling to occur. It also implements vector eligibility which allows the CMAC to control online motion control problems that a normal CMAC would be incapable of controlling in a feedback error learning control configuration.

Most of the above enhancements are applied by exploiting the fact that the CMAC quantizes the input space. However, in this work a continuous version of the IKRLS-CMAC is also developed which is superior in some aspects, and inferior in others when compared to the quantized IKRLS-CMAC. Particularly, it allows the CMAC to be a better function modeler, at the expense of increased computation time and the ability to use credit assignment.

Some gaps in the CMAC literature are also explored, where the lack of this knowledge affects the IKRLS-CMAC. The optimal choice of the generalization parameter is explored, and also some untested claims by previous works are investigated.
The IKRLS-CMAC and the effect of each improvement are tested against some reference third party CMACs on regression, system identification and three simulated online motion control problems. The online motion control problems are the inverted pendulum, a gantry crane and a position controlled inverted pendulum. All tests are performed successfully by the IKRLS-CMACs and are shown to outperform all the third party CMACs.
Acknowledgements

I would like to Acknowledge my supervisors George Coghill and Nitish Patel for their contribution to this work.

I also dedicate this work to my parents Fe and Werner, to my family Angel, Willie, Gemma, Malou, Conrad, Richard, Ria, Rochelle, Zac to my close friends Jeff, James, Emily, Stephanie, Natalie M, Natalie H, Julius, Matthew, Alan and my wonderful partner Linda who supported me all the way...
## Nomenclature and Convention

**Table 1: Nomenclature.**

<table>
<thead>
<tr>
<th>Sym</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>$1 \times n_d$</td>
<td>Raw input vector.</td>
</tr>
<tr>
<td>$i_{\text{min}}$</td>
<td>$1 \times n_d$</td>
<td>Known minimum value for input $i$.</td>
</tr>
<tr>
<td>$i_{\text{max}}$</td>
<td>$1 \times n_d$</td>
<td>Known minimum value for input $i$.</td>
</tr>
<tr>
<td>q</td>
<td>$1 \times n_d$</td>
<td>Quantized input vector.</td>
</tr>
<tr>
<td>n</td>
<td>$1 \times n_d$</td>
<td>Normalized input vector.</td>
</tr>
<tr>
<td>r</td>
<td>$1 \times n_d$</td>
<td>Quantization/Normalization resolution.</td>
</tr>
<tr>
<td>D</td>
<td>$m \times n_d$</td>
<td>Displacement matrix. Defines the type of CMAC overlay used.</td>
</tr>
<tr>
<td>$n_d$</td>
<td>$1 \times 1$</td>
<td>Number of input dimensions.</td>
</tr>
<tr>
<td>d</td>
<td>$n_t \times 1$</td>
<td>Desired output value vector.</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$1 \times 1$</td>
<td>The error given by the difference between the state variable being controlled and the reference value (for online motion control).</td>
</tr>
<tr>
<td>$\varphi$</td>
<td>–</td>
<td>Addressing function.</td>
</tr>
<tr>
<td>$\varphi_{\text{HN}}$</td>
<td>–</td>
<td>Addressing function with higher order basis functions and two-norm normalization.</td>
</tr>
<tr>
<td>x</td>
<td>$1 \times n_w$</td>
<td>Association vector.</td>
</tr>
<tr>
<td>$n_w$</td>
<td>$1 \times 1$</td>
<td>Length of the association vector/feature weight vector.</td>
</tr>
<tr>
<td>W</td>
<td>$n_w \times 1$</td>
<td>Feature space weight vector.</td>
</tr>
<tr>
<td>$\mathbf{w}$</td>
<td>$m \times 1$</td>
<td>The row indices of the activated values in the association vector $\mathbf{x}$.</td>
</tr>
<tr>
<td>h</td>
<td>$1 \times 1$</td>
<td>Local generalization factor. Also, size of a full block.</td>
</tr>
<tr>
<td>m</td>
<td>$1 \times 1$</td>
<td>Number of layers in the CMAC.</td>
</tr>
<tr>
<td>$\mathbf{X}$</td>
<td>$n_\alpha \times n_w$</td>
<td>Dictionary of association vectors.</td>
</tr>
<tr>
<td>k</td>
<td>$n_\alpha \times 1$</td>
<td>Kernel vector.</td>
</tr>
<tr>
<td>K</td>
<td>$1 \times 1$</td>
<td>Kernel peak value.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$n_w \times 1$</td>
<td>Kernel space weight vector.</td>
</tr>
<tr>
<td>$n_\alpha$</td>
<td>$1 \times 1$</td>
<td>Number of weights in the kernel space weight vector. Also, length of the kernel vector.</td>
</tr>
<tr>
<td>$\delta$</td>
<td>$1 \times 1$</td>
<td>Amount by which the current association vector is linearly dependent on the dictionary.</td>
</tr>
<tr>
<td>$\nu$</td>
<td>$1 \times 1$</td>
<td>Sparsification threshold parameter. Controls the number of data added to the dictionary.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Sym</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$n_\alpha \times n_\alpha$</td>
<td>Matrix used for controlling the update strength to $\alpha$ when no dictionary value is added.</td>
</tr>
<tr>
<td>K</td>
<td>$n_\alpha \times n_\alpha$</td>
<td>Kernel matrix. Matrix of kernel vectors which correspond to dictionary values.</td>
</tr>
<tr>
<td>$K^{-1}$</td>
<td>$n_\alpha \times n_\alpha$</td>
<td>Inverse kernel matrix.</td>
</tr>
<tr>
<td>s</td>
<td>$m \times 1$</td>
<td>Sensitivity response vector.</td>
</tr>
<tr>
<td>$o_s$</td>
<td>$1 \times 1$</td>
<td>B-spline sensitivity function order.</td>
</tr>
<tr>
<td>$o_e$</td>
<td>$1 \times 1$</td>
<td>B-spline sensitivity function kernel response order.</td>
</tr>
<tr>
<td>C</td>
<td>$n_w \times n_w$</td>
<td>Credibility matrix. Diagonal matrix used for credit assignment which stores the number of times each feature weight has been used by the dictionary vectors.</td>
</tr>
<tr>
<td>$K_s^{-1}$</td>
<td>$n_\alpha \times n_\alpha$</td>
<td>Credibility updated inverse kernel matrix for credibility method two.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>$1 \times 1$</td>
<td>Credit assignment strength control parameter.</td>
</tr>
<tr>
<td>E</td>
<td>$n_\alpha \times m$</td>
<td>A sparse matrix used to update the kernel weight vector for credit assignment.</td>
</tr>
<tr>
<td>$\xi$</td>
<td>$n_w \times o_e$</td>
<td>The eligibility matrix. This is used to calculate the eligibility association vector and store the current eligibility values.</td>
</tr>
<tr>
<td>$\bar{x}$</td>
<td>$n_w \times 1$</td>
<td>The eligibility association vector.</td>
</tr>
<tr>
<td>$\bar{K}$</td>
<td>$n_\alpha \times n_\alpha$</td>
<td>Recent kernel matrix. Stores recent kernel vectors added to the dictionary in the continuous IKRLS-CMAC.</td>
</tr>
<tr>
<td>$\bar{k}$</td>
<td>$n_\alpha \times 1$</td>
<td>Vector eligibility kernel vector.</td>
</tr>
<tr>
<td>$\omega$</td>
<td>$n_w \times 1$</td>
<td>The decay vector. Each individual element $\omega_i$ keeps track of how many timesteps the decay process for weight $i$ in the feature space has been happening, from the last activation of that weight.</td>
</tr>
<tr>
<td>A</td>
<td>$o_e \times o_e$</td>
<td>System matrix for the discrete linearized model of the plant + feedback controller. Or created from a lower order model as in (6.2).</td>
</tr>
<tr>
<td>b</td>
<td>$o_e \times 1$</td>
<td>System vector for the discrete linearized model of the plant + feedback controller. Or created from a lower order model as in (6.2).</td>
</tr>
<tr>
<td>c</td>
<td>$1 \times o_e$</td>
<td>System vector for the discrete linearized model of the plant + feedback controller. Or created from a lower order model as in (6.2).</td>
</tr>
<tr>
<td>$\psi$</td>
<td>$1 \times 1$</td>
<td>Eligibility decay threshold. Approximate number of timesteps required before the eligibility profile has almost fully decayed to zero.</td>
</tr>
<tr>
<td>$o_e$</td>
<td>$1 \times 1$</td>
<td>Order of the eligibility profile. Larger order allow for more complex eligibility profiles.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>$1 \times 1$</td>
<td>KRLS decay constant. Controls the amount of weight decay in the $P$ matrix when the KRLS-CMAC is used online.</td>
</tr>
<tr>
<td>$g_i$</td>
<td>$1 \times 1$</td>
<td>Initial gain. The more divergent the system, the higher this should be.</td>
</tr>
<tr>
<td>$g_e$</td>
<td>$1 \times 1$</td>
<td>Eligibility gain. Larger values give larger CMAC outputs, leading to larger control force.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Sym</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_d$</td>
<td>$1 \times 1$</td>
<td>Derivative limiting gain. Should be a small percentage of $g_e$. Larger gains reduce the derivative of the CMAC output.</td>
</tr>
</tbody>
</table>

Some conventions that will be used throughout the thesis are defined here.

Table 2: Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold Lower Case</td>
<td>$x$</td>
<td>Indicates a Vector.</td>
</tr>
<tr>
<td>Bold Upper Case</td>
<td>$X$</td>
<td>Indicates a Matrix</td>
</tr>
<tr>
<td>Subscripted Vector</td>
<td>$x_y \leftarrow 1$</td>
<td>A subscripted vector indicates that the vector elements are used as an index into the vector that is being subscripted.</td>
</tr>
<tr>
<td>Subscripted Equation</td>
<td>$x_{y&gt;z} \leftarrow 1$</td>
<td>A subscripted vector with an equation indicates that all vector elements that satisfy the equation will be used as an index.</td>
</tr>
<tr>
<td>Vector/Matrix Reset</td>
<td>$X \leftarrow []$</td>
<td>Assigning $[]$ to a vector or matrix means that it is reset to a $1 \times 1$ element.</td>
</tr>
<tr>
<td>Diagonalizing</td>
<td>diag($x$)</td>
<td>Function diag creates a zero matrix, with the vector elements along the diagonal.</td>
</tr>
<tr>
<td>Matrix Indexing</td>
<td>$X_{i,:}$</td>
<td>This indexing indicates that the row of the matrix is set to $i$, and $:$ indicates that all the columns are selected. So the output is a vector.</td>
</tr>
<tr>
<td>Element Wise Multiplication</td>
<td>$x \circ y$</td>
<td>This indicates elementwise multiplication.</td>
</tr>
</tbody>
</table>
What’s on the DVD

This thesis is accompanied by a DVD which contains the thesis latex files, and a pdf version. The DVD also contains the MATLAB and C# code used in the experiments, and the Excel sheets used to collect and analyze the test data.

MATLAB Programs

- All CMAC implementations, except for the RLS-CMAC.
- Regression test code.
- System Identification test code.
- Motion Control SIMULINK test code.

C# Programs

- The continuous CMAC lookup table brute force generator code.
- RLS-CMAC QR-RLS-CMAC and IQR-RLS-CMAC Test Code.
Contents

List of Figures ........................... xv
List of Tables ............................ xix
Glossary .................................. xxi

1 Review of the CMAC Neural Network .............................. 1
   1.1 Introduction .................................. 1
   1.2 CMAC Model .................................. 1
       1.2.1 $S \rightarrow M$ Mapping ......................... 2
       1.2.2 $M \rightarrow A$ Mapping ......................... 3
       1.2.3 $A \rightarrow P$ Mapping ......................... 3
       1.2.4 CMAC Example ............................... 3
       1.2.5 CMAC Learning ............................. 4
           1.2.5.1 Learning Convergence ................. 5
           1.2.5.2 Weight Initialization .................... 5
       1.2.6 CMAC Weight Space ......................... 5
       1.2.7 Addressing Algorithm ...................... 6
       1.2.8 Hashing ................................ 7
           1.2.8.1 What is Hashing? ..................... 8
           1.2.8.2 Collisions ........................... 8
           1.2.8.3 Hashing in the CMAC .................. 8
       1.2.9 Local Generalization ..................... 9
       1.2.10 Quantizing Resolution ................. 9
       1.2.11 CMAC Algorithm ....................... 10
   1.3 CMAC Applications .......................... 10
   1.4 Feedback Error Learning Control .................. 11
       1.4.1 Other Control Schemes ............... 11
1.5 CMAC Improvements ........................................ 12
  1.5.1 CMAC Memory Handling ................................. 12
    1.5.1.1 Hashing Alternatives ............................ 13
  1.5.2 Recursive Least Squares Training .................... 13
  1.5.3 CMAC Overlays ......................................... 14
    1.5.3.1 Uniform Overlay ................................ 15
    1.5.3.2 Full Overlay and Universal Approximation ....... 15
    1.5.3.3 Full Overlay Approximation ..................... 17
  1.5.4 Learning Interference ................................ 17
    1.5.4.1 Learning Interference Example .................. 18
    1.5.4.2 Learning Interference Solutions ................. 19
  1.5.5 Kernel CMAC .......................................... 20
  1.5.6 Higher Order Sensitivity Functions .................. 21
  1.5.7 Eligibility .......................................... 24
    1.5.7.1 FOX-CMAC .................................. 25
  1.5.8 CMAC Variants not used in this work ................. 26
    1.5.8.1 Fuzzy CMACs .................................. 26
    1.5.8.2 Self Organizing CMACs ........................ 27
    1.5.8.3 Hierarchical CMACs ............................ 27
    1.5.8.4 Multi-Resolution CMACs ......................... 28

1.6 Research Goals .......................................... 29

1.7 Thesis Structure ........................................ 30

2 Inverse QR Decomposition Recursive Least Squares CMAC 33
  2.1 Introduction ............................................ 33
  2.2 QR-RLS-CMAC Introduction ................................ 33
    2.2.1 QR Decomposition ................................ 33
    2.2.2 QR-RLS-CMAC .................................. 34
  2.3 IQR-RLS-CMAC Algorithm ................................ 34
    2.3.1 Algorithm Optimization ............................ 36
  2.4 Parallel IQR-RLS-CMAC ................................ 36
    2.4.1 Parallel Hardware Implementation .................. 37
    2.4.2 Parallel Software Implementation .................. 37
  2.5 Regularization/Credit Assignment for the IQR-RLS-CMAC .... 38
    2.5.1 Optimizations .................................... 43
2.6 Conclusions ......................................................... 43

3 Kernel Recursive Least Squares CMAC .......................... 45
  3.1 Introduction .................................................... 45
  3.2 Introduction to the Kernel Recursive Least Squares Algorithm .... 45
      3.2.1 Brief Introduction to Kernel Methods ..................... 46
          3.2.1.1 General Kernel Example ............................ 47
  3.3 KRLS-CMAC Algorithm ......................................... 48
      3.3.1 CMAC Kernel Advantage .................................. 48
      3.3.2 Online Sparsification ................................... 49
      3.3.3 KRLS Algorithm Modifications .......................... 51
  3.4 KRLS Approximations .......................................... 51
      3.4.1 Approximation to Matrix P .............................. 51
          3.4.1.1 Effect of the Approximation of P .................. 52
  3.5 Conclusions ................................................... 52

4 KRLS-CMAC and Higher Order Sensitivity Functions .......... 55
  4.1 Introduction .................................................. 55
  4.2 Higher Order Sensitivity B-Spline Algorithm ................. 56
      4.2.1 B-Spline Calculation Algorithm ......................... 57
  4.3 Choosing the Generalization Parameter and Overlay Type .... 59
      4.3.1 Local Generalization Area Sizes with Higher Order Sensitivity Functions ......................... 60
      4.3.2 Combination Functions for the Uniform Overlay and Normalization 62
      4.3.3 Association Vector Normalization ........................ 63
      4.3.4 Quantizing Resolution with Higher Order Sensitivity Functions .............................. 64
      4.3.5 Association Vector Algorithm with Higher Order Sensitivity and Normalization .............. 65
  4.4 Experiments .................................................... 65
      4.4.1 Kernel Response Error Function Calculation ............... 65
      4.4.2 Sensitivity Functions Approximation Comparisons .......... 67
          4.4.2.1 Two Dimensional Kernel Response Results .......... 68
          4.4.2.2 Three Dimensional Kernel Response .................. 68
          4.4.2.3 Four Dimensional Kernel Response .................. 69
      4.4.3 Higher Order Sensitivity Resolution Effect .............. 71
      4.4.4 Multidimensional Kernel Contour Error Visualization .... 71
4.4.5 Conclusions ................................................. 74

5 KRLS-CMAC and Credit Assignment ................................................. 75
  5.1 Introduction ................................................. 75
  5.2 Regularization ................................................. 76
    5.2.1 Regularization and Credit Assignment Equivalence ......... 77
    5.2.2 Online KRLS-CMAC Regularization ....................... 77
    5.2.3 Illustration of Kernel Regularization ................... 77
  5.3 Update Equations Derivation ...................................... 78
  5.4 Regularization/Credit Assignment Algorithm ....................... 81
    5.4.1 A Note on Sparse Matrices ............................... 82
    5.4.2 Secondary Credit Assignment Method .................... 82
  5.5 Choosing the Credit Assignment Strength Parameter ............... 84
  5.6 Conclusions .................................................. 86

6 KRLS-CMAC and Vector Eligibility ................................................. 87
  6.1 Introduction .................................................. 87
  6.2 Online Control and Vector Eligibility ............................ 87
    6.2.1 Eligibility Profiles ..................................... 88
  6.3 KRLS-CMAC Eligibility Derivation .................................. 89
    6.3.1 Eligibility Update Equation Derivation .................. 89
    6.3.2 Vector Eligibility Weight Update ........................ 91
    6.3.3 KRLS Eligibility Kernel Weight Decay Parameter ....... 92
    6.3.4 Disturbances to the Reference Signal .................... 93
  6.4 KRLS Vector Eligibility Algorithm ................................... 93
  6.5 Replacing versus Accumulating Eligibility ....................... 93
  6.6 Finding the Eligibility Profile .................................. 95
    6.6.1 Non-Convergent Impulse Responses ....................... 96
  6.7 Conclusion .................................................... 97

7 The Quantized and Continuous KRLS-CMACs ..................................... 99
  7.1 Introduction to the Quantized Improved KRLS-CMAC ............... 99
  7.2 Quantized IKRLS-CMAC Algorithm ................................... 99
  7.3 Abbreviations .................................................. 99
  7.4 Discussion .................................................... 101
  7.5 Minor Improvements to the IKRLS-CMAC ......................... 102
7.5.1 Output Derivative Learning .................................. 102
7.5.2 Initial Gain ............................................... 102
7.6 Introduction to the Continuous Improved KRLS-CMAC .......... 103
7.7 Direct Kernel Vector Calculation ................................ 103
  7.7.1 Quantized Direct Kernel Vector Calculation ................. 103
  7.7.2 Continuous Direct Kernel Vector Calculation ............... 105
  7.7.3 Higher Order Kernels .................................. 105
7.8 Continuous KRLS-CMAC Credit Assignment ..................... 107
  7.8.1 Combinatorics Credit Assignment Algorithm ................. 107
    7.8.1.1 Step 1 - Temporarily Append Current Normalized Input
to the Dictionary ........................................... 108
    7.8.1.2 Step 2 - Calculate the Original Kernel Vector .......... 109
    7.8.1.3 Step 3 - Create $z$ Vector .......................... 109
    7.8.1.4 Step 4 - Find All Combinations of the Values in $z$ .... 109
    7.8.1.5 Step 5 - Select Combinations ........................ 113
    7.8.1.6 Step 6 - Calculate the Unadjusted Subtraction Vector ... 113
    7.8.1.7 Step 7 - Calculate the Subtraction Vector .............. 114
    7.8.1.8 Step 8 - Calculate the Credit Assigned Kernel Value ... 115
    7.8.1.9 Step 9 - Repeat and Finalize ........................ 116
7.8.2 Multidimensional Continuous Credit Assignment ............. 116
7.8.3 Credit Assigned Continuous IKRLS-CMAC Algorithm .......... 116
7.8.4 The Problem with Combinatorics Credit Assignment .......... 118
7.9 Continuous CMAC Vector Eligibility ........................... 118
  7.9.1 Continuous CMAC Vector Eligibility Derivation ............. 118
    7.9.1.1 Continuous Vector Eligibility Example ................ 119
    7.9.2 Continuous Vector Eligibility Implementation .......... 120
    7.9.3 Continuous Eligibility Algorithm ....................... 122
7.10 Advantages and Disadvantages of the IKRLS-CMAC .............. 122
  7.10.1 Gains and Losses for Different Tasks .................... 122
  7.10.2 Cons of IKRLS-CMAC Methods Compared with Other Methods .. 123
7.11 Conclusion ................................................. 124
CONTENTS

8 Regression Tests .............................................. 127
  8.1 Testing Method ........................................... 127
  8.2 Resolution Effects ....................................... 128
  8.3 Minimum vs. Radial and Normalization Sensitivity Function Tests ... 128
    8.3.1 Two Dimensional Results ............................. 129
    8.3.2 Three Dimensional Results ......................... 129
    8.3.3 Four Dimensional Results ......................... 131
  8.4 Comparisons and Correlations with the Kernel Error .............. 132
  8.5 1D Sine Wave Regression Comparison Test ........................ 132
    8.5.1 Results ............................................... 135
  8.6 Multidimensional Sombrero Function Regression Tests ............ 136
  8.7 Conclusions ............................................... 137

9 System Identification Tests .................................. 141
  9.1 Results .................................................. 142
  9.2 Qualitative Comparison Against Other Methods ................. 143
  9.3 Conclusion ............................................... 144

10 Motion Control Test - Inverted Pendulum ................................. 145
  10.1 Results ............................................... 147
    10.1.1 Comparison Error Results .......................... 149
    10.1.2 Average Computation Time Results .................. 150
    10.1.3 Eligibility Decay Value Computation Time Tests ....... 152
      10.1.3.1 Quantized CMAC ................................. 152
      10.1.3.2 Continuous CMAC ............................... 153
    10.1.4 Actual Computation Times ........................... 155
    10.1.5 Eligibility versus no Eligibility CMAC Output Comparisons .. 156
  10.2 Comparison with other CMACs ................................ 156
  10.3 Effect of the Eligibility Profile .......................... 156
  10.4 Conclusion ............................................... 157

11 Motion Control Test - Gantry Crane .................................. 163
  11.1 Results ............................................... 164
    11.1.1 Comparison Error Results ......................... 167
    11.1.2 Average Computation Time Results .................. 167
    11.1.3 Actual Computation Time Results ................... 169
11.2 Conclusion .......................................... 172
12 Motion Control Test - Position Controlled Inverted Pendulum 173
  12.1 Results ........................................... 175
    12.1.1 Comparison Test Results ..................... 175
    12.1.2 Average Computation Time Results .......... 176
    12.1.3 Actual Computation Times ................... 177
  12.2 IKRLS-CMAC Compared against Other Methods on Motion Control Learning ................................ 179
  12.3 Conclusion ...................................... 181
13 Conclusions and Future Work ........................................ 183
  13.1 Main Thesis Contributions ......................... 183
  13.2 Suggested Future Work ............................. 184
  13.3 Conclusions ..................................... 185
A Full Overlay Base Conversion Algorithm ................................ 189
B Introduction to Genetic Algorithms ................................ 191
  1 B.1 Genetic Algorithm Terminology ................... 191
    1 B.1.1 Phenotype ................................... 191
    1 B.1.2 Fitness ..................................... 191
    1 B.1.3 Evaluation .................................. 192
    1 B.1.4 Selection .................................... 192
    1 B.1.5 Reproduction/Crossover ....................... 192
    1 B.1.6 Mutation .................................... 192
  1 B.2 Genetic Algorithm Flowchart ....................... 193
  1 B.3 Advanced Genetic Algorithms ....................... 193
C Calculating the Continuous CMAC Credit Assignment Lookup Tables 195
  1 C.1 Creating the Lookup Tables ....................... 195
    1 C.1.1 Binary Combinations Representation ......... 195
    1 C.1.2 Lookup Table for Combinations Table ....... 197
    1 C.1.3 Lookup Table for Selection Table .......... 197
References .................................................. 199
List of Figures

1.1 CMAC Mapping Overview ........................................... 2
1.2 CMAC Example ...................................................... 4
1.3 CMAC Weight Table Addresses .................................... 7
1.4 Feedback Error Learning ........................................... 12
1.5 Overlays ............................................................. 16
1.6 Learning Interference and Regularization Effect on a Sine Wave .... 18
1.7 Learning Interference Example .................................... 19
1.8 Sensitivity Function Example ..................................... 22
1.9 First Order B-Spline Sensitivity Function CMAC Example ........ 22
1.10 Higher Order B-Spline Output Comparisons ..................... 23
1.11 Eligibility Example ................................................ 24
1.12 Eligibility Effect .................................................. 25
1.13 Eligibility Decay Example ....................................... 26

2.1 Systolic Array ....................................................... 38

3.1 Kernel Classification Illustration ................................ 48

4.1 1D Kernel Response ................................................ 57
4.2 Kernel Contours ..................................................... 61
4.3 Hypercube Topology ............................................... 63
4.4 Kernel Response Comparison ...................................... 63
4.5 2D Kernel Approximation Errors ................................ 69
4.6 3D Kernel Approximation Errors ................................ 70
4.7 4D Kernel Approximation Errors ................................ 70
4.8 3D Overlay Resolution Effect .................................... 72
4.9 Changes over Dimensionality ..................................... 73
# List of Figures

5.1 Kernel Response Credit Assignment Effect ........................................... 78
5.2 Effect of CA Parameter ........................................................................... 86
6.1 Connection Example .................................................................................. 88
6.2 Inverted Pendulum .................................................................................... 92
6.3 Impulse Response with Forced Decay ....................................................... 97
7.1 Credit Assignment Effect ......................................................................... 102
7.2 Kernel Function Insight ............................................................................ 105
7.3 Continuous Credit Assignment Example .................................................. 112
7.4 Tally Visualization ..................................................................................... 114
8.1 2D Combination Function Comparison Tests ........................................... 130
8.2 3D Combination Function Comparison Tests ........................................... 130
8.3 4D Combination Function Comparison Tests ........................................... 131
8.4 2D Regression Correlation ....................................................................... 133
8.5 3D Regression Correlation ....................................................................... 133
8.6 4D Regression Correlation ....................................................................... 134
8.7 4D Regression Correlation ....................................................................... 134
8.8 1D CMAC Ablative Tests .......................................................................... 136
8.9 2D Sombrero Regression Tests ................................................................ 138
8.10 3D Sombrero Regression Tests ............................................................... 138
8.11 4D Sombrero Regression Tests ............................................................... 139
9.1 System Identification Test Results ............................................................ 143
10.1 Simmechanics Inverted Pendulum .......................................................... 146
10.2 Simmechanics Inverted Pendulum Control Scheme .................................. 146
10.3 Inverted Pendulum Impulse Response ...................................................... 148
10.4 Inverted Pendulum Error Results ............................................................ 151
10.5 Inverted Pendulum Computation Times ................................................... 152
10.6 Inverted Pendulum Number of Weights .................................................. 153
10.7 Computation Times for Quantized Eligibility Ablative Tests .................... 154
10.8 Eligibility Decay Continuous CMAC Computation Time Results ........... 154
10.9 Actual Computation Times ...................................................................... 157
10.10 Actual Computation Times ..................................................................... 158
10.11 Eligibility Effect on Pole Angle ............................................................. 158
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.12</td>
<td>Eligibility Effect on Control Effort</td>
<td>159</td>
</tr>
<tr>
<td>10.13</td>
<td>Eligibility Profiles Tested</td>
<td>159</td>
</tr>
<tr>
<td>10.14</td>
<td>Results of Eligibility Profiles Tested</td>
<td>161</td>
</tr>
<tr>
<td>11.1</td>
<td>Simmechanics Gantry Crane</td>
<td>165</td>
</tr>
<tr>
<td>11.2</td>
<td>Simmechanics Crane</td>
<td>166</td>
</tr>
<tr>
<td>11.3</td>
<td>Gantry Crane Impulse Response</td>
<td>166</td>
</tr>
<tr>
<td>11.4</td>
<td>Crane Reference Following Tests</td>
<td>168</td>
</tr>
<tr>
<td>11.5</td>
<td>Gantry Crane Computation Times</td>
<td>169</td>
</tr>
<tr>
<td>11.6</td>
<td>Gantry Crane Weight Usage</td>
<td>170</td>
</tr>
<tr>
<td>11.7</td>
<td>Gantry Crane Actual Computation Times (Weight Not Added)</td>
<td>170</td>
</tr>
<tr>
<td>11.8</td>
<td>Gantry Crane Actual Computation Times (Weight Added)</td>
<td>171</td>
</tr>
<tr>
<td>12.1</td>
<td>Simmechanics Position Controlled Inverted Pendulum</td>
<td>174</td>
</tr>
<tr>
<td>12.2</td>
<td>Position Controlled Inverted Pendulum Impulse Response</td>
<td>175</td>
</tr>
<tr>
<td>12.3</td>
<td>Position Controlled Inverted Pendulum Error Results</td>
<td>176</td>
</tr>
<tr>
<td>12.4</td>
<td>Position Controlled Inverted Pendulum Cart position</td>
<td>177</td>
</tr>
<tr>
<td>12.5</td>
<td>Position Controlled IP Average Computation Times</td>
<td>178</td>
</tr>
<tr>
<td>12.6</td>
<td>Position Controlled IP Number of Weights</td>
<td>178</td>
</tr>
<tr>
<td>12.7</td>
<td>Position Controlled Inverted Pendulum Actual Computation Times (Weight Not Added)</td>
<td>180</td>
</tr>
<tr>
<td>12.8</td>
<td>Position Controlled Inverted Pendulum Actual Computation Times (Weight Added)</td>
<td>180</td>
</tr>
<tr>
<td>1.1</td>
<td>Genetic Algorithm Flowchart</td>
<td>194</td>
</tr>
</tbody>
</table>
List of Tables

1  Nomenclature ................................................................. ii
2  Conventions ................................................................. iv

1.1 Universal Approximation Ratios ........................................ 16

3.1 Approximation to $P$ MSE Results ...................................... 52
3.2 Approximation to $P$ Computation Time Results (in seconds) .... 52

7.1 Combination Table for $n_x = 5$, $n = 4$ and $h = 4$. .............. 111
7.2 Selection Table for $z_1$. .................................................... 113

10.1 Actual Computation Times When $n_\alpha = 32$ ....................... 156

11.1 Maximum Actual Computation Times for the Gantry Crane Experiment
when $n_\alpha = 80$ ................................................................ 172

12.1 Actual Computation Times when $n_\alpha = 127$ ....................... 179

A.1 Full Overlay via Base Conversion for $h = 3$. ......................... 190

C.1 Vector Binary Representation of $z$ ....................................... 196
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Quantized binary sensitivity.</td>
</tr>
<tr>
<td>BE</td>
<td>Quantized binary sensitivity with eligibility.</td>
</tr>
<tr>
<td>C0</td>
<td>Continuous zero order kernel.</td>
</tr>
<tr>
<td>C0E</td>
<td>Continuous zero order kernel with eligibility.</td>
</tr>
<tr>
<td>C3</td>
<td>Continuous third order kernel.</td>
</tr>
<tr>
<td>C3E</td>
<td>Continuous third order kernel with eligibility.</td>
</tr>
<tr>
<td>CA2B</td>
<td>Quantized credit assigned (method 2) binary sensitivity.</td>
</tr>
<tr>
<td>CA2BE</td>
<td>Quantized credit assigned (method 2) binary sensitivity with eligibility.</td>
</tr>
<tr>
<td>CA2L</td>
<td>Quantized credit assigned (method 2) linear sensitivity.</td>
</tr>
<tr>
<td>CA2LE</td>
<td>Quantized credit assigned (method 2) linear sensitivity with eligibility.</td>
</tr>
<tr>
<td>CAB</td>
<td>Quantized credit assigned binary sensitivity.</td>
</tr>
<tr>
<td>CABE</td>
<td>Quantized credit assigned binary sensitivity with eligibility.</td>
</tr>
<tr>
<td>CAL</td>
<td>Quantized credit assigned linear sensitivity.</td>
</tr>
<tr>
<td>CALE</td>
<td>Quantized credit assigned linear sensitivity with eligibility.</td>
</tr>
<tr>
<td>CMAC</td>
<td>Cerebellar Model Articulation Controller.</td>
</tr>
<tr>
<td>FBE</td>
<td>Feedback error learning.</td>
</tr>
<tr>
<td>FOX</td>
<td>Fairly obvious extension (to the CMAC).</td>
</tr>
<tr>
<td>IKRLS</td>
<td>Improved Kernel Recursive Least Squares.</td>
</tr>
<tr>
<td>KLMS</td>
<td>Kernel Least Mean Squares.</td>
</tr>
<tr>
<td>KRLS</td>
<td>Kernel Recursive Least Squares.</td>
</tr>
<tr>
<td>L</td>
<td>Quantized linear sensitivity IKRLS-CMAC.</td>
</tr>
<tr>
<td>LE</td>
<td>Quantized linear sensitivity with eligibility.</td>
</tr>
<tr>
<td>LMS</td>
<td>Least Mean Squares.</td>
</tr>
<tr>
<td>MLP</td>
<td>Multilayer Perceptron.</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional Integral Derivative.</td>
</tr>
<tr>
<td>RLS</td>
<td>Recursive Least Squares.</td>
</tr>
<tr>
<td>SOCM</td>
<td>Self Organizing CMAC.</td>
</tr>
</tbody>
</table>
1

Review of the CMAC Neural Network

1.1 Introduction

This chapter introduces the Cerebellar Model Articulation Controller (CMAC) neural network, and presents a review of improvements found in the literature that have been applied to the CMAC.

The CMAC was created by Albus [1, 2] in 1975, and was originally intended to be a simple biological model [3] of the cerebellar cortex. From medical science we know that the cerebellum is a part of the brain that plays an important role in motor function control, especially in controlling the coordination, precision and timing of movements. Over the years the CMAC has become viewed mostly from an algorithmic perspective, rather than a biological one. This shift in perspective has opened the CMAC to various mathematical interpretations and improvements from the field of machine learning.

The CMAC’s primary advantages over other neural networks are that it learns quickly and is very computationally efficient. Unfortunately, the modeling performance can be poor, but that can be improved through several modifications. In [4] it was suggested that combining multiple CMAC improvements from the literature would be a worthwhile endeavor, and this is the approach taken in this thesis.

Other introductions to the CMAC can be found in [5, 6, 7, 8]. A standard CMAC implementation coded in the C programming language can be found in [9].

1.2 CMAC Model

The CMAC is essentially a look up table with local interpolation capabilities. Mathematically, it can be described as a mapping $S \rightarrow M \rightarrow A \rightarrow P$. This mapping is
1. REVIEW OF THE CMAC NEURAL NETWORK

![Diagram](image)

**Figure 1.1: CMAC Mapping Overview** - Conceptual overview of the CMAC mappings.

shown visually in Fig. 1.1. The mapping takes a real valued input, quantizes it and turns it into a sparse binary vector representation, which is then used to select the corresponding weights. The selected weights are then summed together to produce the output. Each stage of the mapping is explained in detail in the following sections.

1.2.1 $S \rightarrow M$ Mapping

$S \rightarrow M$ is a mapping from an $n_d$ dimensional input vector $i = [i_1 \ i_2 \ \cdots \ i_{n_d}]^\top$ where $i_j \in \mathbb{R}$, to a quantized vector $q = [q_1 \ q_2 \ \cdots \ q_{n_d}]^\top$ where $q_j \in \mathbb{Z}$. Quantization is performed by the function shown in (1.1), where $r$ is the desired quantization resolution vector, and $i_{\text{max}}$ and $i_{\text{min}}$ are vectors of the known bounds of $i$.

\[
q \leftarrow \frac{i - i_{\text{min}}}{i_{\text{max}} - i_{\text{min}}} \tag{1.1}
\]
1.2 CMAC Model

1.2.2 $M \rightarrow A$ Mapping

The mapping $M \rightarrow A$ is a non-linear recoding from vector $\mathbf{q}$ into a higher dimensional binary vector. This binary vector is called the association vector, and is denoted by $\mathbf{x} = [x_1 \ x_2 \ \cdots \ x_{n_w}]^T$ where $x_j \in \{0, 1\}$, and where $n_w$ is the number of weights in the CMAC.

The number of weights in the CMAC can be large, but the association vector will only contain $m$ non-zero '1's, where $m << n_w$. Thus, the association vector is sparse, and can be stored efficiently using any common sparse matrix techniques. The value of $m$ defines the number of layers in the CMAC, which will be explained later in Section 1.2.4. In terms of neural networks, this means that only $m$ weights are activated at any one time, unlike in other neural networks such as the Multilayer Perceptron (MLP) where every weight is activated upon each input excitation.

1.2.3 $A \rightarrow P$ Mapping

In the mapping $A \rightarrow P$ the association vector is used to select and add together $m$ values from an array of weights $\mathbf{w} = [w_1 \ w_2 \ \cdots \ w_{n_w}]^T$ where $w_j \in \mathbb{R}$, to form the output. This can be viewed as an inner product calculation $\mathbf{x}^T \mathbf{w}$. As the CMAC uses quantized inputs, the CMAC output will be piecewise constant and appear staircase-like, where the staircase step size is determined by the quantizing resolution used.

Multidimensional outputs are obtainable simply by using multiple single output CMACs in parallel, where all CMACs receive the same input, but different desired value training data.

1.2.4 CMAC Example

In Fig. 1.2 a commonly used visualization of a two input ($n_d = 2$) CMAC is shown with an example quantized input of $\mathbf{q} = [4 \ 8]^T$, quantizing resolution $\mathbf{r} = 13$ in both dimensions and $n_w = 64$ weights. Here $m = 4$ layers are used, with each layer corresponding to one of the four weight tables on the right of the figure.

It can be seen that the input vector at (4, 8) slices through the four layers on both axes. The sliced letters for each layer activate a certain weight in its corresponding weight table. Each individual weight corresponds to a hypercube in the input space, which for the two dimensional CMAC is simply a square. The activated hypercubes are also referred to as receptive fields, and all the activated hypercubes together form what is known as the local generalization area. The activated hypercubes in the example in
1. REVIEW OF THE CMAC NEURAL NETWORK

Fig. 1.2 are shown as four squares diagonally arranged in the input space. The squares represent the activation of weights $Bc$, $Fg$, $Jk$ and $No$. If put into association vector form it will appear as in (1.2) where $x$ is the association vector, and $\varphi$ is the CMAC addressing function which is discussed later in Section 1.2.7.

$$x^T = \varphi(q) = \begin{bmatrix} A_a & B_a & \ldots & B_c & \ldots & F_g & \ldots & J_k & \ldots & N_o & \ldots & P_g \end{bmatrix}$$

(1.2)

**Figure 1.2:** CMAC Example - A 2D CMAC with $m = 4$ four layers, diagonal overlay, resolution of $r = 13$ and $n_w = 64$ weights.

1.2.5 CMAC Learning

Learning in the CMAC is based on adjusting the value of the activated weights in order to produce a correct output for an input. In the standard CMAC, the LMS algorithm
shown in (1.3) is used for this purpose, where \( \mu \) is the learning rate and \( e \) is the training error. The training error is either obtained through (1.4), where \( d_t \) is the desired target value for input \( t \), or by directly using an externally generated error, which will act to move the weights in the general direction of error minimization.

\[
\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{old}} - \frac{\mu}{m} \mathbf{x} e \\
e = x^T \mathbf{w} - d_t
\]  

(1.3)  

(1.4)

The association vector used in (1.3) ensures that only the activated weights that are selected by the association vector will be updated.

The work in [10] shows a method using genetic algorithms to automatically select an optimal learning rate, however this has the obvious flaw of being an offline only solution.

1.2.5.1 Learning Convergence

In [11] it is shown that as long as the learning rate is set in the range \( 0 < \mu < 2 \), the CMAC will at least converge to a limit cycle. It is further shown that if \( 0 < \mu < 1 \), the error is guaranteed to reduce with training. Thus it is recommended the learning rate be kept between these bounds.

1.2.5.2 Weight Initialization

Unlike the MLP where the weight vector is initialized with randomized values, the CMAC weights are initialized to zero. This can be advantageous, especially for online learning, as untrained areas of the input space will simply output zero and not a random value. This can prevent an online control system from receiving a possibly detrimental or damaging input.

1.2.6 CMAC Weight Space

In the standard CMAC, the number of layers is given by \( m = h \), where \( h \) is known as the generalization parameter, and is defined by the chosen size of a ‘full block’ in the CMAC. Examples of full blocks are shown in Fig. 1.2, where blocks A, B, C on dimension one layer one are all full blocks, as they span the maximum distance of \( h = 4 \) over the quantized input space. Note that later it will be seen that \( m \) does not need to be equal to \( h \).
1. REVIEW OF THE CMAC NEURAL NETWORK

The number of weights in the CMAC can be calculated with (1.5), where \( D \) is a matrix that specifies the amount of displacement a particular layer \( i \), in dimension \( j \) uses. For example, in Fig. 1.2 the first layer has a displacement of zero, and the second layer has a displacement of one. Different displacement arrangements are possible and are discussed in Section 1.5.3. Also, \( r_j \) is the resolution for dimension \( j \).

\[
    n_w = \sum_{i=1}^{m} \prod_{j=1}^{n_d} \left[ \frac{r_j + D_{i,j}}{h} \right] \tag{1.5}
\]

As can be seen from (1.5), the number of weights required can increase dramatically when the input dimensionality, quantizing resolution or number of layers, \( m \) are large.

1.2.7 Addressing Algorithm

The addressing algorithm shown in Algorithm 1 is used to convert a quantized input \( q \) into \( m \) activated addresses that can be used to create an association vector. Other material related to the addressing algorithm can be found in [6] and [9].

In a CMAC there are always \( m \) weight tables. To find the activated address for the first weight table, the dimensions of the weight table must first be found. This can be found through use of (1.6) which creates a temporary variable storing the dimension lengths, where \( i \) represents the weight table of interest, and \( j \) represents the dimension of interest, where here dimension one is defined as the bottom dimension in Fig. 1.2.

\[
    p_{i,j} \leftarrow \left[ \frac{r_j + D_{i,j}}{h} \right] \tag{1.6}
\]

Next, the index of the activated block for each dimension must be found using (1.7).

\[
    t_{i,j} \leftarrow \left[ \frac{q_j + D_{i,j}}{h} \right] \tag{1.7}
\]

Now the weight table indices can be found using (1.6) and (1.7) in (1.8), where \( \bar{w} \) is a vector of weight table addresses.

\[
    \bar{w}_l \leftarrow t_{i,1}p_{i,1}^0 + t_{i,2}p_{i,2}^1 + \ldots + t_{i,n_d}p_{i,n_d}^{n_d-1} + \sum_{j=1}^{n_d} \prod_{k=1}^{i} \left[ \frac{r_k + D_{j,k}}{h} \right] \tag{1.8}
\]

In Algorithm 1 the variable \( v \) is used to sum the total number of weights used in previous weight tables to the address. The returned vector \( \bar{w} \) contains the \( m \) weight table addresses of the activated weights which can be used to create the sparse association vector.
Figure 1.3: CMAC Weight Table Addresses - How the weight table addresses are laid out for the CMAC example. Each number is the weight table address and the shaded values are the activated weights in the example.

Note that the operation $x_w \leftarrow 1$ indicates that each value in $x$ addressed by the addresses stored in $\tilde{w}$ are set to unity. Weight table addresses activated in the CMAC example are shown in Fig. 1.3. Weight indices 6, 25, 41 and 57 are activated. So, $x_6$, $x_{25}$, $x_{41}$ and $x_{57}$ will all be set to unity.

The addressing algorithm may involve a secondary step for hashing. This is not shown here, but is explored in Section 1.2.8 and in [6].

Algorithm 1 CMAC Addressing Algorithm

1: procedure $\varphi(q)$
2: for $i \leftarrow 1 : m$ do
3: $p \leftarrow 1$
4: $v \leftarrow 0$
5: for $j \leftarrow 1 : n_d$ do
6: $\tilde{w}_i \leftarrow \tilde{w}_i + \left[ \frac{q_j + D_t e_j}{h} \right] p$
7: $p \leftarrow p \left[ \frac{r_j + D_t e_j}{h} \right]$
8: end for
9: $\tilde{w}_i \leftarrow \tilde{w}_i + v$
10: $v \leftarrow v + p$
11: end for
12: $x_w \leftarrow 1$
13: return $[x \tilde{w}]$
14: end procedure

1.2.8 Hashing

As seen by the number of weights that can be calculated with (1.5), the memory requirements of the CMAC can be potentially very large. In the simple example given
in the previous sections, only 64 weights are required. However, the number of weights required grows significantly with increasing resolution and input dimensionality. A typical CMAC with four inputs, a generalization parameter of $h = 30$, and a quantizing resolution of 300 will use 434589 weights. As it is unlikely that every single weight in the CMAC will be used, hash mapping can be used to compress this memory requirement down to a more manageable size.

1.2.8.1 What is Hashing?

Hashing is a technique often used in computer science [12]. The idea is to map a large virtual address space down to a much smaller physical address space. This is done with a hash function which calculates the mapping from virtual to physical. A good hash function will pseudo-randomly distribute the virtual addresses amongst the physical addresses, avoiding any sort of pattern.

1.2.8.2 Collisions

Obviously, there cannot be a one to one mapping between the large number of virtual and small number of physical addresses, and so several virtual addresses will map to the same physical address. When two virtual weights that map to the same physical address are used, a collision occurs. When this happens the old weight stored in the physical memory will be corrupted by the current update which was intended for another virtual weight.

1.2.8.3 Hashing in the CMAC

In the CMAC collisions will be rare if it is assumed that only a small area of the input space will be explored, as this means that the majority of virtual weights will remain unused, thus ensuring collisions are statistically unlikely. The number of collisions can be further reduced by ensuring that the hash function maps the virtual addresses to the physical addresses in a pseudo-random manner. Also, due to local generalization, several neighboring virtual weights will be activated during a single training cycle. If the hash function is good, each activated virtual weight will map to a unique physical weight.

The standard CMAC uses a form of hash mapping where collisions are simply ignored, and the resulting collision errors are viewed as a noise source at the CMAC output. The CMAC is able to tolerate collision errors well since the CMAC output
1.2 CMAC Model

consists of the sum of several activated weights, thus effectively averaging the output. A single incorrect weight due to a collision will be negligible if the other \( m - 1 \) activated weights are correct.

The motivation for using hashing in the CMAC is further explored in [13].

1.2.9 Local Generalization

Unlike other neural networks which have global generalization, such as the Multilayer Perceptron (MLP), the CMAC has local generalization. With local generalization only the activated weights are adjusted instead of all the weights. This is an advantage as new learning will not disrupt previous learning that was done in a far away region of the input space.

The CMAC generalization ability is governed by a single parameter [14] which is defined in this work as \( \phi \). It is the ratio shown in (1.9) that determines the local generalization area size.

\[
\phi = \frac{h}{r}
\]  

(1.9)

This is because for a fixed value of \( h \), a lower resolution will cause the hypercubes to spread over a larger area in the input space. If the resolution needs to be increased, the resolution and the value of \( h \) should be increased together in a way such that the ratio of \( \phi \) stays the same so the local generalization area size does not change. Note that both \( h \) and \( r \) must be integer.

Increasing the generalization ratio increases the local generalization area size in the CMAC, improving results when data is sparse in the input space, since the CMAC will be able to interpolate between those data. But large local generalization area sizes restrict CMAC modeling accuracy if there is tightly spaced data, as subtle variations in the data will be smoothed. Therefore, if data is sparse a larger generalization ratio should be used for greater interpolation ranges, but when there is sufficient data, smaller generalization ratios should be used to improve the modeling accuracy.

1.2.10 Quantizing Resolution

Choosing a quantization resolution too small will cause multiple real valued inputs to map to the same quantized value, and thus share the same set of weights. This will cause old weights to be corrupted by new weights. This is especially problematic in areas of the input space that require modeling finesse as the subtle details will be lost.
Algorithm 2 CMAC Algorithm

1: procedure CMAC(i, d)
2:     for t ← 1, 2…nt do
3:         q ← Quantize(i_t)
4:         \[ \begin{bmatrix} x \\ \hat{w} \end{bmatrix} \leftarrow \varphi(q) \]
5:         y ← x^\top w
6:         w ← w − \mu \frac{1}{m} x (d_t − y)
7:     return y
8: end for
9: end procedure

Increasing the resolution increases the number of weights and spreads the quantized input out more, making it less likely that weights will be shared.

The quantizing resolution can be different for each dimension, but usually it is set as the same over all dimensions. The quantizing resolution should be increased by increasing h and r such that the same value of \( \phi \) is maintained.

1.2.11 CMAC Algorithm

The standard CMAC algorithm is shown in Algorithm 2. In this algorithm the input is first quantized and turned into an association vector through the addressing algorithm shown in Algorithm 1. The CMAC output is then calculated through an inner product using the association vector and weight vector, and then the weight vector is updated through the LMS iterative algorithm. To just get the output, the LMS update algorithm in line 6 can be omitted. This algorithm ignores hashing, which would be incorporated into the addressing function if used.

1.3 CMAC Applications

The original application of the CMAC was the control of robotic manipulators [15]. It has since found its place in many other applications, and in the following a brief cross section of applications found in the literature is presented.

The CMAC has found applications in simulated ship steering [16], medical imaging [17], control of proton exchange membrane fuel cells [18], artificial muscle control [4], general data classification [19, 20], ECG classification [21], diagnosis of liver disease [22], synchronization of chaotic gyro systems [23], control of chaotic systems [24, 25], control
of induction motors [26], control of a quadrocopter [27], and blood glucose regulation for diabetics [28].

1.4 Feedback Error Learning Control

In control systems a standard control approach is to use both a feedback and feedforward controller together. The feedforward controller provides the main control, and the feedback controller handles any inaccuracies in the feedforward model, and also any disturbances or noise. The feedforward controller can be derived through mathematical modeling of the system, or it can be a neural network that learns the forward model online or offline.

Feedback error learning uses the feedback control signal as an error signal, which is used to train a neural network based feedforward controller in real time. This method was first described in [15, 29], and later in [30]. It has successfully been used for a number of years in applications such as robotic arm control [15], automatic car braking [31] and underwater robot thruster control [32].

In Fig. 1.4 a typical feedback error learning setup is shown, where the CMAC is used as the feedforward controller. This figure shows that the feedback error control signal is connected to the CMAC and used as the training error $e$ in the LMS algorithm. Over time the feedback controller ‘teaches’ the CMAC the correct output for a particular desired system state feedforward input denoted by $y_d$. The main control influence will at first be the feedback controller, but eventually the feedforward CMAC controller will become more influential as it learns the control function. It must be noted that after training, the feedback controller cannot be removed as the CMAC like any other neural network is unlikely to ever learn a perfect model. Also a feedforward controller is unable to handle disturbances or reject noise. Typically the feedback controller is a commonly used proportional integral derivative (PID) controller.

Instead of using the desired system state as a feedforward input, it is also valid to use the current measured system state $u$ as the input to the CMAC. This method is usually used when trying to stabilize a system to some sort of optimal reference, rather than control its trajectory.

1.4.1 Other Control Schemes

There are also other ways in which the CMAC can be used to control a system. Apart from feedback error learning, the most common is a combination of sliding mode control
1. REVIEW OF THE CMAC NEURAL NETWORK

![CMAC Diagram]

**Figure 1.4: Feedback Error Learning** - Example of feedback error learning with the CMAC as the feedforward learning controller.

and lyapunov control theory. There are various examples in the literature where this control scheme is used [33]. In [25] a self-organizing CMAC is combined with sliding mode control to control a double inverted pendulum.

The problem with these control methods is that they usually require some prior knowledge of the system in the form of a mathematical model. For some systems, modeling of the system may be too difficult or impossible.

Reinforcement learning is also another common control area in which the CMAC has proven to be useful. For example in [34] a robotic manipulator is controlled by a CMAC trained with reinforcement learning, and in [35] the CMAC is used in an adaptive critic learning problem.

In this work, these other control schemes are kept out of the scope of the project, as the CMAC built in this work is suited only for use in feedback error learning due to the use of vector eligibility. There may be a way to combine vector eligibility with the other CMAC control schemes, and this may be interesting future work.

1.5 CMAC Improvements

The literature provides a multitude of improvements to the standard Albus CMAC which are used to enhance the CMACs capabilities and fix certain flaws. These flaws and their improvements are introduced and discussed in the following sections.

1.5.1 CMAC Memory Handling

With modern computer hardware, the amount of memory the CMAC requires is usually not so troubling. In most cases it is now acceptable to directly access memory
without any form of hashing as modern computer memory is plentiful. However, in some cases when the CMAC is very high dimensional, or the full overlay is used (see Section 1.5.3.2), the total amount of memory required may larger than what is currently feasible.

Alternative approaches to basic hashing that are collision free may be used in today’s fast hardware, and some methods previously used in the CMAC are described in the following.

1.5.1.1 Hashing Alternatives

In [9] collision-resistant hashing is implemented, which is a hashing method that simply assigns extra memory when the first collision occurs, instead of overwriting the data. Other common completely collision free hashing techniques that dynamically allocate memory in chunks such as linear hashing [36, 37] can be used, but these collision free techniques come at the expense of increased computation time and increased memory use.

In [7] it is stated that binary search trees (BSTs) are a suitable alternative to hashing in the CMAC as they allow efficient collision free operation with dynamic memory allocation. However, BSTs do not seem to provide much benefit over collision free hashing techniques as they use roughly the same amount of memory for overheads, but are also slower at storing and accessing data. Additionally, the BST requires a tree balancing algorithm to be performed when the CMAC is taken offline in order to ensure optimal computational performance.

In [38] a direct weight mapping approach, which is able to utilize memory efficiently is implemented into hardware. In [39, 40] a content addressable memory is used as the CMAC memory, which is a hardware implementation of the typical key-value pair associative hash table based dictionary used in most programming languages.

1.5.2 Recursive Least Squares Training

As discussed in the CMAC introduction, the CMAC uses the LMS algorithm to train the weights. An alternative training algorithm used in [41] is the Recursive Least Squares (RLS) algorithm. The RLS algorithm has the advantage that it will converge after just one presentation of the training set, and that it does not require the tuning of a learning rate. Its much faster convergence rate makes it desirable for use in online
1. REVIEW OF THE CMAC NEURAL NETWORK

motion control as explained in [41]. The CMAC with RLS training algorithm will be referred to as the RLS-CMAC in this thesis.

Unfortunately for RLS, the price paid for these improvements is a computational complexity of $O(n_w^2)$. And as was shown earlier, the number of weights in the CMAC $n_w$ can be massive for even modest input space dimensionalities. Thus this renders the RLS based CMAC too computationally demanding for input spaces with dimensionalities greater than around two. In [42] an RLS algorithm based on QR-decomposition [43] (QR-RLS) is used in the CMAC (called the QR-RLS-CMAC), which improves computation speed and allows for efficient hardware implementation on a systolic array, and also increases numerical stability. However the computational complexity remains unchanged. The work in [42] optimizes the QR-RLS algorithm for the CMAC, but does so only for one dimensional problems, and still does not reduce the computational complexity.

A method developed by the author of this thesis to improve the computational speed of the QR-RLS-CMAC is presented in Chapter 2, and is based on the authors previous work in [44]. It uses an inverse QR-RLS method to avoid a computationally costly back substitution matrix operation which is a major bottle neck in the QR-RLS-CMAC. Additionally, in Chapter 2, credit assignment (to be introduced later in this chapter) is implemented into the IQR-RLS-CMAC improving the modeling properties.

However, although the IQR-RLS-CMAC improves the computational speed of the RLS algorithm by reducing the number of operations, the final complexity still remains the same at $O(n_w^2)$. This makes it still unable to be useful for real time online learning without hardware implementation. The RLS-CMAC shows that a different type of algorithm is required for real time online RLS-CMAC learning.

1.5.3 CMAC Overlays

In the CMAC the way the letters in the layers are arranged determines the way the hypercubes will be arranged in the input space. This arrangement of the hypercubes is referred to as the overlay and is controlled by matrix $D$. In Fig. 1.2 a CMAC with a diagonal overlay is used and this is also shown in Fig. 1.5a. The diagonal overlay is the simplest overlay as each successive layer is simply displaced by one quantization unit to the left of the previous layer. The uniform overlay is illustrated in Fig. 1.5b and significantly improves upon the diagonal overlay by spacing the hypercubes in a way such that the input space is covered more evenly.
1.5.3.1 Uniform Overlay

In [45] the uniform overlay displacements are calculated through an exhaustive search, and in [46] they are calculated through use of a genetic algorithm as the exhaustive search becomes too computationally demanding for high dimensionalities or a large number of layers $m$. However, in [9] a simple heuristic algorithm is used, which can easily calculate uniform overlay displacements that are almost equivalent to the exhaustive search for any dimensionality and value of $m$.

1.5.3.2 Full Overlay and Universal Approximation

It is now a well known fact that the standard multidimensional CMAC is not a universal approximator in the sense that it cannot exactly reproduce its own training set like a lookup table can [47, 48], unlike the univariate CMAC. In a univariate CMAC, the number of weights is always greater than or equal to the number of possible input states, and is thus able to represent any quantized function presented. This can be thought of in terms of solving simultaneous equations where there are more variables than equations, thus allowing many solutions.

Multidimensional CMACs keep the same number of layers as a univariate CMAC (since $m = h$), resulting in fewer weights than there are possible input states. Thus, there are less variables than equations giving an overdetermined system with no solutions, meaning that the multidimensional CMAC will be unable to exactly represent each input state. The CMAC training algorithm will however find the least squares solution. This problem increases with larger CMAC dimensionalities as the ratio of number of input states to weights increases.

For example, take a CMAC with $r = 300$, $h = m = 30$ with the uniform overlay. In Table 1.1 the number of feature space weights used by the CMAC, the number of possible quantized states and the resulting ratio between the two is shown. The table clearly shows that as the dimensionality increases the ratio between the number of possible states and number of weights quickly increases.

Increasing the value of $h$ whilst keeping $\phi$ constant actually increases the universal approximation ratio in multidimensional CMACs, since the number of states increases faster than the number of feature space weights with larger $h$ due to the required larger resolution. However, in a typical CMAC system, it can be assumed that the number of states actually visited will be much less than all the states available. Thus, since the number of states actually visited is constant, the ratio between the number of states...
1. REVIEW OF THE CMAC NEURAL NETWORK

Table 1.1: Universal Approximation Ratios

<table>
<thead>
<tr>
<th>#D</th>
<th>( n_w ) Full</th>
<th>( n_w ) Uniform</th>
<th># States</th>
<th>Uniform Ratio</th>
<th>Full Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>330</td>
<td>330</td>
<td>300</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>2D</td>
<td>108900</td>
<td>3630</td>
<td>300(^2)</td>
<td>25</td>
<td>0.83</td>
</tr>
<tr>
<td>3D</td>
<td>35937000</td>
<td>39930</td>
<td>300(^3)</td>
<td>676</td>
<td>0.75</td>
</tr>
<tr>
<td>4D</td>
<td>( 1.1859 \times 10^{10} )</td>
<td>439230</td>
<td>300(^4)</td>
<td>18441</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Figure 1.5: Overlays - The a) diagonal b) uniform and c) full overlays.

actually visited and the number of feature space weights will actually improve with larger \( h \) and constant \( \phi \).

A complete solution to this universal approximation problem is to use the full overlay which is illustrated in Fig. 1.5c. If the full overlay is used, the CMAC gains many more feature space weights and becomes a universal approximator with the ability to exactly recall the training set just like the univariate CMAC. The full overlay increases the number of layers in the CMAC to the amount needed to fully cover the local generalization area such that each quantized input will map to a unique set of weights. The uniform overlay approximation ratios for the full overlay can be seen in Table 1.1. The number of layers required for the full overlay is a function of the generalization parameter and the input dimensionality. It is shown in (1.10).

\[
m = h^n_d
\]  

(1.10)

The full overlay displacements for matrix \( D \) are easily calculated by performing a number base conversion that converts the layer \( i \) value into base \( h \). The resulting conversion is then padded to \( n_d \) figures. A base conversion algorithm for calculating \( D \) for the full overlay is shown in Appendix B.

Use of the full overlay is difficult in practical applications as it causes the CMAC
1.5 CMAC Improvements

to require significantly more weights, as can be seen by substituting (1.10) into (1.5). Furthermore, significantly more calculation is required since the number of weights to be updated depends on $m$, which will be large for high dimensional problems.

1.5.3.3 Full Overlay Approximation

The uniform overlay approximates the full overlay with a reduced number of hypercubes. The approximation quality is governed by the size of the generalization parameter $h$. A large $h$ means more hypercubes and thus more building blocks to build the full overlay approximation. Using a higher order basis function which is discussed later in Chapter 4 can also improve the approximation significantly. In Chapter 4, the uniform overlay approximation to the full overlay error for various generalization parameter values is quantified.

1.5.4 Learning Interference

The CMAC learns best when each training sample is spaced such that the local generalization areas that each quantized input activates do not overlap each other [48]. This is because if two training samples are very close to each other, their local generalization areas will overlap and weights will be shared. Sharing weights should not normally be a problem in a neural network, however in [48] it is shown that the CMAC is naturally poor at optimally distributing error corrections over shared weights. This poor error correction distribution is termed ‘learning interference’, since the problem occurs because new learning interferes with old learning through overlapping local generalization areas.

Learning interference is caused by the fact that there will be certain weights in overlapping regions that are activated more frequently than others because they are shared. These frequently activated weights will grow to much larger values as they are activated often, compared with those updated less frequently whose weight values will remain small to compensate for the large frequently activated weights. This unevenness causes unwanted dips in the CMAC output such as those shown in Fig. 1.6a, where a sine wave is trained on a CMAC with training data samples spaced at 30 degree intervals, and parameters set as $r = 200, h = 20$. Its output is sampled at one degree intervals.

In [48] it was shown that the spacing of the training samples and the generalization parameter chosen will decide the amount of learning interference produced. The authors
1. REVIEW OF THE CMAC NEURAL NETWORK

Figure 1.6: Learning Interference and Regularization Effect on a Sine Wave - (a) The effect of learning interference on a sine wave model and (b) the effect of regularization on the same sine wave model.

...showed that for a fixed distance between quantized training samples denoted by $q_d$, the ratio $\frac{h}{q_d}$ will determine the amount of learning interference. Notably, no learning interference will occur if the learning interference ratio is integer. While these results show a way to reduce learning interference, they are only valid for evenly spaced data. In a real world application, it is unlikely training samples will be so well behaved. Further results also showed that the larger the learning interference ratio, the lower the learning interference for any quantized training sample spacing. However, this implies a larger value of $h$ and thus a larger generalization ratio, meaning more smoothing and potentially greater modeling error.

1.5.4.1 Learning Interference Example

Figure 1.7 shows an example of learning interference for a standard LMS based one dimensional CMAC. The local generalization parameter is set to $h = 3$, and learning rate $\mu = 1$. Initially, the first quantized input, $q_1 = 5$ which has a desired value of $d_1 = 12$ is applied to the CMAC. According to the LMS equation in (1.3), the weights are then adjusted by distributing the error over the local generalization area, as can be seen in Fig. 1.7a. Next, $q_2 = 7$ and $d_2 = 15$ is applied in Fig. 1.7b and the activated weights are adjusted accordingly. Now it can be seen in Fig. 1.7b that learning at $q_1$ has been corrupted by the learning done at $q_2$, as one weight was shared between the two inputs due to overlapping receptive fields. The desired output for $q_1$ was 12, but now it is $4 + 4 + 7.67 = 15.67$. If $q_2 = 8$, training would be perfect as there would be no receptive field overlaps, since the learning interference ratio would be $\frac{3}{3} = 1$, which is integer. However, ensuring the training points are uniformly spaced at the correct distance is usually not possible when training online or with random data. In
1.5 CMAC Improvements

![Image](125x709 to 213x718)

**Figure 1.7: Learning Interference Example** - Learning interference illustrated in a one dimensional example. (a) Effect of first training input, (b) Effect of second training input.

some cases continued training may eventually correct the learning interference, but this increases the amount of training required.

1.5.4.2 Learning Interference Solutions

Learning interference can be reduced through several solutions that are found in the literature. One solution is neighborhood sequential and random training [49], which simply adjusts the presentation of the training set so that it is less likely that the same weights will be continually activated. However, this technique can not be used for real time training as the entire data set must be available before training. Optimized weight smoothing [50] and regularization [48] both work by adding a constraint to the error function that forces the activated weights to remain similar to one another in value. This prevents the overlapping weights growing too large and dominating the output. A modified LMS algorithm can then be derived with those constraints applied. Figure 1.6b shows the sine wave example with regularization applied. Tikhonov training [51], works by forcing smoothness through use of output derivative constraint minimization in the error function. Tikhonov training can be thought of as a generalization of optimized weight smoothing and regularization. Credit assignment [52, 53] is a solution almost identical to regularization except that it was born from intuition, rather than a mathematical derivation. The idea is to distribute errors in proportion to the inverse of a hypercube’s credibility, where credibility is defined by the number of times the hypercube has been activated. Thus less error correction is given to frequently activated weights preventing them from growing too large. The mathematics of credit assignment work out to be very similar to regularization. In [54] a credit assigned CMAC is trained
with a genetic algorithm, but this has the flaw of being an offline only system. Weight smoothing [6] is similar to optimized weight smoothing, regularization and credit assignment as it works by adjusting each weight so that it does not deviate significantly from the average of all the activated weights. However, with weight smoothing this adjustment occurs after the presentation of a training sample, and is not built into the update function.

1.5.5 Kernel CMAC

In the literature many kernel based learning algorithms exist, the most notable being the support vector machine [55]. In [56] the kernel LMS algorithm is introduced as a kernelized least mean squares algorithm.

In a kernel machine, the input vector is non-linearly transformed into a high dimensional ‘feature vector’ by a kernel function. The work in [48] introduces what we term the kernel least mean squares CMAC (KLMS-CMAC) which makes the connection that the CMAC can be implemented as a kernel machine. The $M \rightarrow A$ mapping to the association vector is a non-linear transformation to a higher dimension where the kernel used is a zero order binary b-spline function. Using this knowledge, a common method used in kernel machines called the ‘kernel trick’ can be applied. This allows the weights to be evaluated in the ‘kernel space’ rather than the feature space. Since the dimensionality of the kernel space is equal to the size of a dictionary defined by $n_a$, which sparsely stores previously admitted feature vectors (association vectors in the CMAC), significantly less memory is required for weight storage. The kernel vector used in the KLMS-CMAC is given by (1.11) where $\mathbf{x}$ is the current association vector in column form and $\mathbf{X}$ is the dictionary which is a matrix of previously admitted association vectors stored in row form.

$$\mathbf{k} = \mathbf{Xx} \quad (1.11)$$

Use of the kernel vector can be seen later in Chapter 3. In the KLMS-CMAC, the LMS algorithm is used to train the weights in the kernel space. This is shown in (1.12) where $\alpha$ is the kernel space weight vector, $\mu$ is the learning rate, $\mathbf{K}$ is a matrix of kernel vectors for each dictionary entry, and $\mathbf{e}$ is a vector of errors for each training sample.

$$\alpha \leftarrow \alpha + 2\mu \mathbf{Ke} \quad (1.12)$$
The work in [57] applies hash encoding methods to the association vector used in the kernel equation in order to reduce the complexity of the kernel vector calculation. When combined with regularization the noise produced by hash collisions was reduced.

The creators of the KLMS-CMAC suggest use of a sparsification technique, but no sparsification method for the KLMS-CMAC is shown in [48]. This means that all training data presented to the CMAC will be added to the dictionary, and thus the dictionary will grow without bound.

### 1.5.6 Higher Order Sensitivity Functions

The output of the standard CMAC is discontinuous due to the quantization of the input and the resulting binary association vector. The sensitivity function of a CMAC hypercube defines the output of a hypercube when activated by an input. A standard binary hypercube has a sensitivity function that always responds with a ‘1’ when activated. An obvious improvement to the CMAC is to modify the hypercube sensitivity functions so that hypercubes that are activated near the center respond with a larger value than those activated near the edges. An example of this is shown in Fig. 1.8b, where the darkly shaded hypercubes have been activated near their center, and the lightly shaded hypercubes have been activated near their edges. Compare this with Fig. 1.8a where each hypercube is evenly activated with the binary sensitivity function.

Another example that illustrates how higher order basis functions are applied in the CMAC is shown in Fig. 1.9. In this figure a one dimensional CMAC is shown with a first order b-spline sensitivity function. The hypercubes respond with a value based on the intersection of the first order b-spline (triangle function), rather than simply with the constant value ‘1’. For example, a normalized input of $n = 2$ would respond with blocks $\{ A = 1, E = 0.5, J = 0, N = 0.5 \}$. Also, an example input value of $n = 2.2$ will respond with different values $\{ A = 0.9, E = 0.4, J = 0.1, N = 0.6 \}$. Those response values were calculated using simple trigonometry. With the first order b-spline used as the sensitivity function, the input values of 2 and 2.2 are differentiated from one another, whereas with the binary sensitivity function they would be equivalent.

In [58] it was shown that the binary receptive field sensitivity functions can be equivalently represented with a zero order b-spline. It then follows that higher order b-splines can also be used as sensitivity functions. Higher order b-splines are continuous and not quantized, so they create a smooth CMAC output. The b-spline sensitivity functions are first implemented into the CMAC addressing function in [58].
1. REVIEW OF THE CMAC NEURAL NETWORK

Figure 1.8: Sensitivity Function Example - An example of sensitivity weightings on CMAC hypercubes. Darker indicates a greater weighting. a) Standard binary sensitivity function b) Linear sensitivity function.

Figure 1.9: First Order B-Spline Sensitivity Function CMAC Example - A one dimensional CMAC with a first order b-spline sensitivity function.
1.5 CMAC Improvements

![CMAC function graphs](image)

**Figure 1.10: Higher Order B-Spline Output Comparisons** - Comparisons between (a) low res, binary (b) low res, first order b-spline (c) low res, second order b-spline (d) high res, binary (e) high res, first order b-spline and (f) high res, second order b-spline.

Other sensitivity functions such as the Gaussian function [59, 60] have been used in the CMAC with good results as well. In [9] the linear sensitivity function is used, which is equivalent to a first order b-spline. In [61] the Parametric-CMAC (P-CMAC) is introduced where the sensitivity functions used are parametric fuzzy equations. In [62] wavelet functions are used as the sensitivity function.

When using higher order b-splines, the CMAC real valued input vector must be normalized, rather than quantized. Implementation of normalization is simple, just remove the floor function in the quantization function shown in (1.1).

Besides smoother outputs, another advantage to using higher order b-splines is that lower normalization resolutions can be used than what would have been required with the binary sensitivity function. In Fig. 1.10 comparisons between sine wave models for binary, first and second order b-spline sensitivity functions are shown. Graphs (a), (b) and (c) show the sine wave model for a low resolution CMAC ($r = 50, h = 5, \phi = 0.1$), and graphs (d), (e) and (f) show a CMAC with double the resolution ($r = 100, h = 10, \phi = 0.1$). The output of the low resolution CMAC is poor for the binary and first order b-spline, but the second order b-spline gives a smooth output. For the higher resolution CMAC the outputs are similar for the first and second order b-splines. In conclusion, this shows that higher order b-splines can be used to overcome some of the problems with rough outputs common in CMACs with low quantizing resolutions.
1. REVIEW OF THE CMAC NEURAL NETWORK

![Diagram of CMAC with input trajectory and key for weight eligibility]

**Figure 1.11: Eligibility Example** - Exponentially decaying eligibility example shown on a two dimensional CMAC quantized input space. Image sourced from [6].

### 1.5.7 Eligibility

Eligibility is a concept that is used to improve the CMACs ability to control real time motion control problems through learning better control models and learning to control the system faster. In a CMAC with eligibility [35, 63, 64], each individual weight is associated with a scalar called an eligibility value. When a weight is activated, the eligibility value for that weight increases, and when that weight is not activated the eligibility value decays slightly. Eventually the eligibility value will decay to zero if its corresponding weight is not activated for some time. The idea is that all the weights (even non-active weights) should be updated with the current error, which will be scaled to be in proportion to the weight’s eligibility value. It may seem that updating all the weights could be computationally intensive, however a majority of the weights do not need to be updated because their eligibility values will be zero due to decay. Eligibility is illustrated in Fig. 1.11 on a two dimensional CMAC quantized input space, where the eligibility value magnitude is represented by the shadings. Darker shades indicate weights that are more eligible for change as they have been used more recently in time, and lighter shades indicate weights that have had their eligibility values decayed significantly.

Apart from learning to control a system faster, a preempting effect is introduced by eligibility in the CMAC and is illustrated in Fig. 1.12. In this figure a CMAC is trained to control an actuator’s position with a step change. As no actuator can follow a discontinuous step trajectory exactly, the measured output when using a standard CMAC will appear similar to curve two in the figure. Without eligibility the CMAC is only able to preempt the trajectory change only very slightly due to local generalization. This leads to a large difference between the desired step change and actual actuator output. When eligibility is used however, the CMAC is able to minimize the error in
1.5 CMAC Improvements

![Diagram showing trajectory effects]

**Figure 1.12: Eligibility Effect** - A CMAC with eligibility has a preempting effect which allows for better control of an actuator step change. Image sourced from [6].

the step change much more by learning to begin the actuator movement before the step change occurs. Obviously this is a useful feature for motion control problems.

1.5.7.1 **FOX-CMAC**

When a CMAC is used in a feedback error learning configuration (CMAC + FBE) [30], it has three limitations that make it difficult to use for controlling online systems. Firstly, in [6] it was shown that a CMAC + FBE configuration is only able to properly control systems that have a zero order impulse response; that is, a system where the effect of an input to the system is immediately observable in the output states. Secondly, the CMAC + FBE configuration has the requirement that the reference trajectory must be continuous. And thirdly, there is a requirement that the reference trajectory must always begin at the systems initial state.

The Fairly Obvious eXtension to the CMAC (FOX-CMAC) is fully derived in [6] and is a modification to the standard CMAC in which a concept named ‘vector eligibility’ is implemented. The FOX-CMAC is able to overcome the three aforementioned limitations of the CMAC + FBE configuration, and also improves control performance significantly.

With standard eligibility, the eligibility value decays typically through an exponential decay profile that is usually tuned through trial and error. Vector eligibility on the other hand allows for more complex eligibility decay profiles, which can provide an eligibility response that matches the system dynamics more accurately. In order to obtain this complex eligibility profile, the impulse response of the system must be mea-
sured, and thus the system must have a well defined impulse response in the first place. An example of the difference between standard and vector eligibility decay profiles is illustrated in Fig. 1.13. In this image the more complex eligibility profile shown in Fig. 1.13b is generated through vector eligibility equations.

The FOX-CMAC has been used to successfully control an inverted pendulum, gantry crane, line following robot, hopping robot, and bipedal walking robot [6]. The work in [65] uses the FOX-CMAC combined with fuzzified learning to control a simulated ship steering problem.

The work in [4] creates a vector eligibility-like effect by using a second CMAC which adaptively learns to accurately preempt discontinuous changes in the control signal (termed situations) like in Fig. 1.12, but without actually using eligibility. The advantage of the method in [4] is that no tuning of the eligibility profile is required, and no impulse response needs to be measured like with the FOX-CMAC, however the general improvement in controllability and learning speed for non-discontinuous changes is not present.

1.5.8 CMAC Variants not used in this work

1.5.8.1 Fuzzy CMACs
Fuzzy CMACs make use of fuzzy logic theory in order to enhance CMAC modeling, and allow for human interpretable networks through fuzzy rules. A comprehensive overview of fuzzy CMACs is given in [66]. In [67] a Fuzzy-CMAC system is formulated
where the CMAC is used to generate fuzzy inference rules through training. Another similar fuzzified CMAC is shown in [68]. In [69] a recurrent fuzzy CMAC is used in hierarchical form for dynamic system identification. In [70] a hierarchical approach is taken to fuzzy CMACs in order to reduce multidimensional memory usage. In [71] a fuzzy CMAC is combined with credit assignment to obtain better results. In [72] it is proven that a fuzzy system with b-spline membership functions combined with a CMAC can universally approximate an arbitrarily smooth function. In [73] a fuzzy CMAC which first runs a clustering method to determine the receptive field centers is presented.

Fuzzy CMACs are very similar to higher order basis functions, as they both produce an improvement by causing the output to be smoothed. The main difference is that the trained Fuzzy CMACs are much more able to be easily interpreted by humans.

1.5.8.2 Self Organizing CMACs

In [25, 74] a self-organizing CMAC is presented. Self-organization is described as the ability to grow or prune the layers of the CMAC. However, these self-organizing CMACs are actually essentially KLMS-CMACs, where the self-organizing part is based on a gradient descent algorithm that optimizes the dictionary entries. The authors of [25] actually provide a more advanced solution than the basic KLMS-CMAC, and call it the Self Organizing CMAC (SOCM). The SOCM is able to automatically adjust the individual kernel widths and centers as well as the weights via use of the LMS algorithm. The SOCM also uses a sparsification technique (unlike the KLMS-CMAC) to manage the dictionary size by growing and pruning it based on a min-max technique.

In [75] a self-organizing CMAC is described which incorporates ideas from the Kohonen self-organizing map [76] for sparsification, and also uses ideas from 'gray relational analysis' to speed up training.

1.5.8.3 Hierarchical CMACs

Hierarchical CMACs work by using multiple CMACs in hierarchy to simplify the job of each individual CMAC. In [77] the CMAC is used in a hierarchical mixture of experts architecture which works by breaking a control problem down into sub-problems, where each sub-problem is handled by a single CMAC.

Another hierarchical CMAC approach is to break down the modeling problem by dimension in order to solve a high dimensional problem without using too many weights.
1. REVIEW OF THE CMAC NEURAL NETWORK

This is the approach taken in the HCMAC [78, 79] which uses a combination of two dimensional CMACs in a layered hierarchy, where the CMAC outputs connect to the inputs of other CMACs in the hierarchy. The work in [80, 81] also uses two dimensional CMACs in a layered hierarchy, but uses the product of the CMAC outputs rather than connecting them to other CMAC layers. The Macro Structure CMAC (MS-CMAC) [82] also works in a similar way by decomposing the problem into a tree structure of one dimensional CMACs.

1.5.8.4 Multi-Resolution CMACs

In [83] a multi-resolution based CMAC is implemented. In a multi-resolution CMAC multiple generalization lattices are used, where each lattice is an input space with a different generalization parameter. The idea is that the lattices with large local generalization areas will provide good interpolation for areas where the input data is sparse, and the lattices with finer local generalizing areas will provide greater detail where the input data is dense. The work in [83] works by first training the large generalization area lattices, then sequentially training the finer lattices with the difference between the already trained lattice CMAC outputs and the desired value. The work in [84] extends upon the multi-resolution CMAC by modifying it to work with online time varying functions. In [85] a similar approach is used where the generalization size of the lattice assigned to a training data point is based on variance of the desired values. Highly varying areas of the data will be assigned a smaller generalization area. Additionally, unlike in [83], only the coarsest lattice spans the entire input space, where the finer lattices only span areas of the input space that require better resolution. In [86] multiple CMACs of differing resolutions are trained together and their outputs averaged to form the output. The work in [87] combined two CMACs together with different generalization parameters. The two CMACs were used to solve the generalization-accuracy trade-off when controlling an inverted pendulum. In [88] the hierarchically clustered adaptive quantization CMAC is shown. This CMAC uses hierarchical clustering for the quantization of the input space in order to identify regions of the input space where more resolution is required. In [89] yet another approach is described which uses a continuous encoding to adaptively alter the CMAC topology by learning the optimal positions for the CMAC layers.
1.6 Research Goals

In the thesis found in [4], it is suggested that creating a CMAC with many improvements combined may be a worthwhile endeavor. Previous works have worked on adding singular improvements into the CMAC, such as those described in the previous sections. But these CMACs have failed to combine and investigate the effects of combining multiple improvements. In this thesis several compatible improvements are chosen and combined together to form two versions of a new improved CMAC. The new CMACs are then investigated and tested through multiple well known experiments.

The goal of this work is to find a way to successfully combine the chosen improvements together. We will begin by combining each improvement with the KRLS-CMAC, and then finally putting them all together. A continuous non-quantized version of the KRLS-CMAC will also be investigated, and determined if some of the improvements can be applied to it. A list of research goals are presented below.

1. **Implement a Recursive Least Squares CMAC** - The goal is to make the CMAC more useful in online control (and also offline learning) by improving learning and convergence speed.

2. **Implement a Kernel CMAC** - The goal is to mitigate the computational inefficiencies of the recursive least squares algorithm by reducing the weight vector size.

3. **Implement Higher Order Basis Functions into the Kernel CMAC** - The goal is to remove the modeling inefficiencies of quantization by allowing continuous inputs and thus producing smooth outputs.

4. **Investigate Improved Overlays** - The goal is to bring the CMAC closer to acting like it is using the full overlay by using a more optimal overlay.

5. **Implement Regularization/Credit Assignment in the Kernel CMAC** - The goal is to eliminate or reduce the learning interference flaw in the Kernel RLS-CMAC.

6. **Implement Vector Eligibility into the Kernel CMAC** - To improve the online motion control capabilities of the CMAC.
1. REVIEW OF THE CMAC NEURAL NETWORK

7. **Find out if it is possible to use all the improvements together** - Not all improvements are compatible with one another. The improvements mentioned in the goals above were chosen as they are believed to be compatible with one another.

8. **Find out if a continuous non-quantized CMAC with the above improvements is possible** - It may be possible to create a continuous non-quantized CMAC with the above improvements that does not use an association vector.

The improved CMAC will be called the Improved Kernel Recursive Least Squares CMAC and will be referred to as the IKRLS-CMAC henceforth. The second implementation of the IKRLS-CMAC will be one with the same improvements, but will not use an association vector or any form of quantization. This CMAC will be referred to as the continuous IKRLS-CMAC.

Hashing improvements are not required in the IKRLS-CMAC, as the weight space is significantly shrunk due to the kernelization of the CMAC. Self organization is not implemented, as it is difficult to implement in the KRLS algorithm without significantly increasing computational complexity. Hierarchical and multi-resolution improvements can still potentially be applied to the IKRLS-CMAC, however they are avoided here for simplicity and left as potential future work as we feel the improvement they give is not significant if proper CMAC generalization and resolution parameters are chosen.

1.7 Thesis Structure

This thesis is presented as follows. This chapter provided an overview of the CMAC neural network, and showcased a number of CMAC enhancements found in the academic literature. It also laid out the expected goals of this work. In Chapter 2, the Inverse QR-RLS-CMAC is introduced, which is a computational speed improvement over the standard RLS-CMAC and QR-RLS-CMACs found in the literature. It will be shown that no matter the computational speed savings, the computational complexity of RLS based CMACs will not change. In Chapter 3 the Kernel RLS-CMAC will be introduced as a way to get around the computational complexity issues of the RLS-CMAC. Chapter 4 investigates the effect of higher order basis functions in the KRLS-CMAC, and makes some discoveries on the generalization parameter choices. In Chapter 5, credit assignment/regularization is applied to the KRLS-CMAC, and in Chapter 6 vector eligibility is implemented. In Chapter 7 all the enhancements explored
in the previous chapters are combined into a single CMAC algorithm. Additionally, a continuous version of the KRLS-CMAC with some enhancements is investigated and implemented. In Chapter 8 the improved KRLS-CMACs are tested on a regression problem, and in Chapter 9 they are tested on a system identification problem. In Chapters 10, 11 and 12 the improved KRLS-CMACs are tested on online motion control learning problems such as a simulated inverted pendulum, simulated gantry crane and simulated position controlled inverted pendulum respectively. Finally, in Chapter 13 some potential future work is outlined, and conclusions are drawn.
1. REVIEW OF THE CMAC NEURAL NETWORK
2

Inverse QR Decomposition
Recursive Least Squares CMAC

2.1 Introduction

This chapter introduces the inverse QR decomposition based RLS-CMAC (IQR-RLS-CMAC) which is an improvement over the standard RLS-CMAC [41] and the QR-RLS-CMAC [42] in terms of computational speed. The inverse QR-RLS decomposition algorithm is described in [90, 91], and avoids a costly matrix back substitution step required in the QR-RLS algorithm for calculating the weights.

Although the inverse QR decomposition RLS algorithm is not used in the final IKRLS-CMAC, it is a useful development that may be practical in some applications, as it provides a natural parallel hardware implementation of the RLS-CMAC. This chapter also illustrates the fact that the RLS-CMAC will always be computationally inefficient on a single CPU for a large number of weights due its $O(n_\text{w}^2)$ computational complexity which cannot be avoided.

2.2 QR-RLS-CMAC Introduction

In this section the QR-RLS-CMAC which was first implemented in [42] will be briefly introduced.

2.2.1 QR Decomposition

QR-decomposition [90] is a mathematical tool used for decomposing a matrix into two matrix components, where one matrix is orthogonal and denoted by $Q$, and the other
is an upper triangular matrix denoted by $R$. This is demonstrated in (2.1).

$$A = QR \quad (2.1)$$

RLS algorithms tend to suffer from poor numerical stability which can often cause problems. QR decomposition can be applied to the RLS algorithm in order to improve numerical stability [90]. However, the trade off is that the QR-RLS algorithm is slightly more computationally intensive.

QR-decomposition can be computed either by the Gram-Schmidt Process [90], Householder transforms [90] or Givens rotations [90]. In this work the Givens rotation method is used as it is used successfully previously in [42] for the QR-RLS-CMAC. It is also the most computationally efficient method.

2.2.2 QR-RLS-CMAC

The work in [42] tailors the QR-RLS algorithm specifically for the CMAC, resulting in successfully halving the computation time for a one dimensional problem. Unfortunately, the tailored algorithm is only suitable for univariate CMACs as the authors of [42] assume that the association vector uses a method where the activated ‘1’ s are contiguous, which cannot be the case for a multivariate CMAC.

Another problem with QR decomposition is that a costly matrix back substitution step of $O\left(n^2\right)$ complexity must be carried out each time a training sample is presented in order to calculate the weight vector. This back substitution step can be avoided entirely by using an inverse QR-RLS algorithm, which instead allows the weights to be calculated directly.

2.3 IQR-RLS-CMAC Algorithm

In [43] the IQR-RLS algorithm is derived, and here in Algorithm 3, the IQR-RLS algorithm is implemented into the CMAC to create the IQR-RLS-CMAC. This algorithm is also discussed in the author of this thesis’s previous work [44]. This algorithm uses the Givens rotation method to perform the QR-decomposition which is presented as a function in Algorithm 4. Additionally some computational enhancements that are specific to the CMAC implementation are used.

Note that $\delta$ is a constant that is usually set between 10 and 10000. Larger values of $\delta$ theoretically give better results, though it was found that setting $\delta$ too large causes floating point inaccuracies. A value of 100 was found to work well in almost all cases.
Algorithm 3 IQR-RLS-CMAC Algorithm
1: procedure IQR-RLS-CMAC(i, d)  
2: \( w \leftarrow 0, \ x \leftarrow 0, \ R^{-T} \leftarrow \delta I(\delta >> 1), \ \rho \leftarrow 0.001 \)
3: for \( t \leftarrow 1, 2 \ldots n_t \) do
4: \( q \leftarrow \text{Quantize}(i_t) \)
5: \( \begin{bmatrix} x & \bar{w} \end{bmatrix} \leftarrow \varphi(q) \)
6: \( a \leftarrow R^{-T}x \)
7: \( U \leftarrow 0 \)
8: \( \alpha \leftarrow 1 \)
9: \( b \leftarrow \bar{w}_1 \)
10: for \( i \leftarrow b : 1 : n_w \) do
11: \( \begin{bmatrix} R & U & \alpha \end{bmatrix} \leftarrow \text{Givens}(R, U, \alpha, a, i) \)
12: end for
13: \( y \leftarrow x^\top w \)
14: \( e \leftarrow d - y \)
15: \( z \leftarrow \frac{e}{\alpha_{n_w}} \)
16: \( w \leftarrow w - zU_{n_w} \)
17: return \( y \)
18: end for
19: end procedure

Algorithm 4 Givens Rotation Algorithm
1: procedure GIVENS(R, U, \( \alpha, a, i \))
2: if \(|a_i| > \rho\) then
3: \( \alpha_i \leftarrow \sqrt{\alpha_{i-1}^2 + a_i^2} \)
4: \( s \leftarrow \frac{a_i}{\alpha_i} \)
5: \( c \leftarrow \frac{\alpha_{i-1}}{\alpha_i} \)
6: for \( j \leftarrow 1 \ldots i \) do
7: \( U_{i,j} \leftarrow cU_{i-1,j} + sR_{i,j} \)
8: \( R_{i,j} \leftarrow cR_{i,j} - sU_{i-1,j} \)
9: end for
10: end if
11: return \( \begin{bmatrix} R & U & \alpha \end{bmatrix} \)
12: end procedure
2. INVERSE QR DECOMPOSITION RECURSIVE LEAST SQUARES CMAC

2.3.1 Algorithm Optimization

There are three computational speed enhancements that are implemented in Algorithm 3 and Algorithm 4. The first enhancement involves line 9 of Algorithm 3. In line 9 the address where the first '1' appears in the association vector is stored in variable b. The for loop in line 10 then begins its computation from this address. This is because the vector calculated in line 6 will consist of all zeros until the address of the first '1' in \( x \), because of the fact that \( R^{-\top} \) is lower triangular. If \( a_i \) is zero, then in the Givens macro algorithm, value \( s \) will equal zero, and value \( c \) will equal one, resulting in no change for \( R_{i,j} \) and \( U_{i,j} \), rendering any calculations performed redundant.

The second improvement follows on from this where the calculation of lines 3 - 7 in Algorithm 4 are gated by line 2, and thus are only performed if the absolute value of \( a_i \) is greater than \( \rho \) which is set to a small value just above zero. Values of \( a_i \) are often zero due to the sparseness of the CMAC input and the sparseness of the association vector, which leaves \( R^{-\top} \) sparse. The variable \( \rho \) is set to be slightly larger than zero. This is because during the matrix-vector multiplication in line 6, values are often added and subtracted to form the sum of zero, but due to floating point inaccuracies the result will not equal exactly zero, hence the threshold. Generally, a value for \( \rho \) between 0.000001 and 0.001 worked well in almost all cases.

The final computational improvement is that a sparse matrix-vector multiplication can be performed in line 6 of Algorithm 3 because the association vector is sparse, and the addresses of the '1's are known from the \( \bar{w} \) array. Thus, only \( m \) values for each row of \( R^{-\top} \) need to be added.

It can be seen from line 10 of Algorithm 3 that computation time will increase with an increase in the number of weights \( n_w \) required by the CMAC, with a computational complexity given by \( O\left(\frac{n_w^2}{2}\right) \). This is a slight improvement over the \( O\left(n_w^2\right) \) complexity of the RLS-CMAC.

2.4 Parallel IQR-RLS-CMAC

As the RLS-CMAC, QRRRLS-CMAC and IQR-RLS-CMAC all have high computational complexities, it would be desirable to be able to run their calculations in parallel. In this section it is shown how to parallelize the IQR-RLS-CMAC on a systolic array and on a multi-core PC processor.

36
2.4 Parallel IQR-RLS-CMAC

2.4.1 Parallel Hardware Implementation

When Givens rotation is used in the QR-RLS algorithm, it is naturally and optimally parallelized on a systolic array as is done in [42]. The IQR-RLS algorithm from Algorithm 3 can also be parallelized on a systolic array in a similar and even simpler way. A systolic array implementation of IQR-RLS is shown in Fig. 2.1. In this figure, each circle represents a processing element that stores the values of $R^{-\top}$. The circles in the left most column implement lines 3 - 7, and all other circular processing elements implement lines 8 and 7 in Algorithm 4. The square boxes calculate the weights using lines 14 - 16 in Algorithm 3.

2.4.2 Parallel Software Implementation

Often access to systolic array hardware is not available, and only general purpose CPUs are. Parallelization on a PC may be performed by emulating the systolic array computation structure with threading. Here a method is proposed for parallelizing the IQR-RLS algorithm on a PC with a multi-core CPU.

In a visual sense using Fig. 2.1, Algorithm 3 sequentially updates the $U_{i,j}$ and $R_{i,j}$ values in the systolic array row by row. It is, however, equally valid to update the $U_{i,j}$ and $R_{i,j}$ values column by column instead. Since each column is independent from one another in terms of value dependencies (apart from $c$ and $s$), it is possible to update each column simultaneously without creating memory sharing bottlenecks. A description on how the IQR-RLS-CMAC may be parallelized is described in two steps below.

**Step 1** - The first step that needs to be performed is sequential in nature. First, realize that the $c$ and $s$ values are constant across each row. Thus, the values of $c$ and $s$ must first be sequentially calculated for each row and stored in an array. The value $\alpha_{n,w}$ is also calculated as a by-product from the calculations and is stored for later use. This is a fast computation.

**Step 2** - In step two, realize that $U_{i,j}$ and $R_{i,j}$ can be updated column by column. Since each column is independent from one another, each column can be updated in a separate thread, one for each core on the CPU. Further optimization can be achieved by combining computation of shorter columns together to equalize thread computation times, and by skipping any calculations on rows where the $a_i$ value is below the threshold $\rho$. 

37
2. INVERSE QR DECOMPOSITION RECURSIVE LEAST SQUARES CMAC

Figure 2.1: Systolic Array - Parallel systolic array implementation of the IQR-RLS-CMAC algorithm.

2.5 Regularization/Credit Assignment for the IQR-RLS-CMAC

Here the regularization concept is applied to the IQR-RLS algorithm in order to combat the learning interference problem. Regularization is introduced more in depth in Chapter 5. A partial mathematical derivation is presented below based on the IQR-RLS derivations found in [43]. First, from [48] see the least squares cost function given by $\varepsilon(t)$ in (2.2), where $t$ is the total number of training samples.

$$
\varepsilon(t) = \sum_{i=1}^{t} \left( \left( d(i) - x(i)^{\top} w(t) \right)^2 + \lambda \sum_{j=1}^{m} \left( \frac{d(i)}{m} - w_j(t) \right)^2 \right)
$$

(2.2)

The first term in (2.2) is the error between the desired and actual CMAC output. The second is the regularization term which adds to the cost the more that the activated weights are different to one another. The constant $\lambda$ is used to control the regularization strength, and it was found setting it to the reciprocal of $\delta$ generally worked best. In order to solve this cost function, (2.2) must be written in vector-matrix form. Define vectors $w$, $d$, $h$ and matrices $X$, $\Sigma$ as in (2.3), (2.4), (2.8), (2.5) and (2.9) respectively, where $s$ and $G$ are given by (2.7) and (2.6).
\[ w(t) = \begin{bmatrix} w_1(t) \\ w_2(t) \\ \vdots \\ w_n(t) \end{bmatrix}_{n_w \times 1} \quad (2.3) \]

\[ s(t) = \begin{bmatrix} d(t) \\ \vdots \\ d(t) \end{bmatrix}_{n_w \times 1} \quad (2.7) \]

\[ d(t) = \begin{bmatrix} d(1) \\ d(2) \\ \vdots \\ d(n_w) \end{bmatrix}_{n_w \times 1} \quad (2.4) \]

\[ h(t) = \begin{bmatrix} G(1)s(1) \\ G(2)s(2) \\ \vdots \\ G(t)s(t) \end{bmatrix}_{n_w \times 1} \quad (2.8) \]

\[ X(t) = \begin{bmatrix} x^\top(1) \\ x^\top(2) \\ \vdots \\ x^\top(t) \end{bmatrix}_{t \times n_w} \quad (2.5) \]

\[ \Sigma(t) = \begin{bmatrix} G(1) \\ G(2) \\ \vdots \\ G(t) \end{bmatrix}_{n_w \times n_w} \quad (2.9) \]

Note that \( \text{diag}(x(t)) \) creates a zero matrix with the entries of \( x(t) \) along the main diagonal. Using (2.3) - (2.9) the cost function in (2.2) can be rewritten in matrix-vector form as (2.10).

\[ \epsilon(t) = \| d(t) - X(t)w(t) \|^\top + \lambda \| h(t) - \Sigma(t)w(t) \|^\top \quad (2.10) \]

Where \( \| a \| = \sqrt{a^\top a} \). Equation (2.10) can then be rewritten as a single term by first defining matrix \( A(t) \) and vector \( y(t) \) as in (2.11) and (2.12).

\[ A(t) = \begin{bmatrix} X(t) \\ \sqrt{\lambda} \Sigma(t) \end{bmatrix}_{(t+n_w) \times n_w} \quad (2.11) \]

\[ y(t) = \begin{bmatrix} d(t) \\ \sqrt{\lambda}h(t) \end{bmatrix}_{(t+n_w) \times 1} \quad (2.12) \]

Then, using (2.11) and (2.12) to rewrite \( \epsilon(t) \) gives (2.13).

\[ \epsilon(t) = \| y(t) - A(t)w(t) \|^\top \quad (2.13) \]

Matrix \( A(t) \) is of dimension \( (t+n_w) \times n_w \), and according to [43], there exists a \( (t+n_w) \times (t+n_w) \) orthogonal matrix \( Q(t) \) such that (2.14) holds true.

\[ Q(t)A(t) = \begin{bmatrix} R(t) \\ 0 \end{bmatrix} \quad (2.14) \]

Where \( R(t) \) is the \( n_w \times n_w \) upper triangular Cholesky factor, and \( 0 \) is an \( (t+n_w) - 39 \)
2. INVERSE QR DECOMPOSITION RECURSIVE LEAST SQUARES CMAC

\[ Q(t)y(t) = \begin{bmatrix} z(t) \\ v(t) \end{bmatrix} \] (2.15)

Where \( \mathbf{z}(t) \) is a \( n_w \times 1 \) vector, and \( \mathbf{v}(t) \) is a \((t + n_w t) - n_w \) \( \times 1 \) vector. Since \( Q(t) \) is orthogonal, pre-multiplying each term in (2.13) with \( Q(t) \) does not change the value of the norm, and thus the cost function may be rewritten as (2.16).

\[ \epsilon(t) = \|Q(t)y(t) - Q(t)A(t)w(t)\|\] (2.16)

Finally, substituting (2.14) and (2.15) into (2.16) gives the desired equivalent form shown in (2.17).

\[ \epsilon(t) = \left\| \begin{bmatrix} \mathbf{z}(t) - R(t)\mathbf{w}(t) \\ \mathbf{v}(t) \end{bmatrix} \right\| \] (2.17)

It can then be seen that the norm in (2.17) will be minimized if (2.18) is true.

\[ R(t)\mathbf{w}(t) = \mathbf{z}(t) \] (2.18)

Using (2.18) the weights can be solved for with back substitution. However, for IQR-RLS the \( R^{-1}(t) \) matrix is required so that back substitution can be avoided, so the derivation must continue.

Now the problem becomes how to update \( R(t-1) \) to \( R(t) \) and \( \mathbf{z}(t-1) \) to \( \mathbf{z}(t) \). First, consider the non-regularized solution. In the non-regularized solution, \( A(t) \) and \( y(t) \) in (2.13) are replaced with \( X(t) \) and \( d(t) \). In [43] it is shown that an \((n_w + 1) \times (n_w + 1)\) orthogonal matrix \( T(t) \) exists that will perform the non-regularized update by updating using the latest entry of \( X(t) \) and \( d(t) \), which are \( x(t) \) and \( d(t) \) respectively as is shown in (2.19) and (2.20).

\[ T(t) \begin{bmatrix} R(t-1) \\ x^\top(t) \end{bmatrix} = \begin{bmatrix} R(t) \\ 0^\top \end{bmatrix} \] (2.19)

\[ T(t) \begin{bmatrix} z(t-1) \\ d(t) \end{bmatrix} = \begin{bmatrix} z(t) \\ \xi(t) \end{bmatrix} \] (2.20)

However, with regularization, (2.13) uses \( A(t) \) and \( y(t) \) which is a composition of two matrices and two vectors respectively, so it must be updated with the latest entries of \( X(t) \), \( d(t) \) and the latest entries of \( \sqrt{\lambda} \Sigma(t) \), \( \sqrt{\lambda} \mathbf{h}(t) \) which are \( \sqrt{\lambda} \mathbf{G}(t) \) and \( \sqrt{\lambda} \mathbf{G}(t)s(t) \) respectively. There must then exist an \((2n_w + 1) \times (2n_w + 1)\) matrix \( T(t) \) such that
(2.21) is true.

\[
T(t) \begin{bmatrix}
R(t - 1) \\
\mathbf{x}^\top(t) \\
\sqrt{\lambda} \mathbf{G}(t)
\end{bmatrix} = \begin{bmatrix}
R(t) \\
0^\top \\
0
\end{bmatrix}
\]  
(2.21)

And similarly to update \( \mathbf{z}(t - 1) \) to \( \mathbf{z}(t) \), use (2.22).

\[
T(t) \begin{bmatrix}
\mathbf{z}(t - 1) \\
d(t) \\
\sqrt{\lambda} \mathbf{G}(t) \mathbf{s}(t)
\end{bmatrix} = \begin{bmatrix}
\mathbf{z}(t) \\
\xi(t) \\
\varphi(t)
\end{bmatrix}
\]  
(2.22)

Unfortunately, if (2.21) and (2.22) are used, the same derivations found in [43] cannot be used, as the derivations are suited only to a \((\mathbf{n}_w + 1) \times (\mathbf{n}_w + 1)\) matrix. However, realize that \( \mathbf{R}(t) \) may be calculated iteratively if a second iteration count function parameter is defined, and it is defined that \( \mathbf{R}(t, \mathbf{n}_w) \equiv \mathbf{R}(t), \mathbf{z}(t, \mathbf{n}_w) \equiv \mathbf{z}(t) \) and \( \mathbf{G}(t) \) and \( \mathbf{s}(t) \) are written row by row as in (2.23) and (2.24).

\[
\mathbf{G}(t) = \begin{bmatrix}
\mathbf{g}_1(t) \\
\mathbf{g}_2(t) \\
\vdots \\
\mathbf{g}_{\mathbf{n}_w}(t)
\end{bmatrix}
\]  
(2.23)

\[
\mathbf{G}(t) \mathbf{s}(t) = \begin{bmatrix}
\mathbf{g}_1(t) \mathbf{s}(t) \\
\mathbf{g}_2(t) \mathbf{s}(t) \\
\vdots \\
\mathbf{g}_{\mathbf{n}_w}(t) \mathbf{s}(t)
\end{bmatrix}
\]  
(2.24)

Then first update as usual using \( \mathbf{x}(t) \) and \( d(t) \) as in (2.25) and (2.26), just like in [43].

\[
T(t, 0) \begin{bmatrix}
\mathbf{R}(t - 1) \\
\mathbf{x}^\top(t)
\end{bmatrix} = \begin{bmatrix}
\mathbf{R}(t, 0) \\
0^\top
\end{bmatrix}
\]  
(2.25)

\[
T(t, 0) \begin{bmatrix}
\mathbf{z}(t) \\
d(t)
\end{bmatrix} = \begin{bmatrix}
\mathbf{z}(t, 0) \\
\xi(t)
\end{bmatrix}
\]  
(2.26)

Now, using \( \mathbf{g}_1(t) \) to \( \mathbf{g}_{\mathbf{n}_w}(t) \) iteratively update until \( \mathbf{R}(t, \mathbf{n}_w) \) is calculated as shown in (2.27) and (2.28).

\[
T(t, 1) \begin{bmatrix}
\mathbf{R}(t, 0) \\
\sqrt{\lambda} \mathbf{g}_1(t)
\end{bmatrix} = \begin{bmatrix}
\mathbf{R}(t, 1) \\
0^\top
\end{bmatrix}
\]  
(2.27)

\[
\vdots
\]

\[
T(t, \mathbf{n}_w) \begin{bmatrix}
\mathbf{R}(t, \mathbf{n}_w - 1) \\
\sqrt{\lambda} \mathbf{g}_{\mathbf{n}_w}(t)
\end{bmatrix} = \begin{bmatrix}
\mathbf{R}(t, \mathbf{n}_w) \\
0^\top
\end{bmatrix}
\]  
(2.28)

Next, using \( \mathbf{g}_1(t) \mathbf{s}(t) \) to \( \mathbf{g}_{\mathbf{n}_w}(t) \mathbf{s}(t) \) iteratively update until \( \mathbf{z}(t, \mathbf{n}_w) \) is calculated as in
2. INVERSE QR DECOMPOSITION RECURSIVE LEAST SQUARES CMAC

Algorithm 5 Regularized IQR-RLS algorithm for the CMAC

1. First perform one run of Algorithm 3, but replace line 16 with line 2 of this algorithm.
2. \( v \leftarrow z u \)
3. Then while still inside the main training sample loop
4. \( \lambda \leftarrow \frac{1}{\delta} \)
5. \( G \leftarrow \text{diag}(x(t)) \)
6. for \( j \leftarrow 1, 2 \ldots m \) do
7. \( p \leftarrow \bar{w}_j \)
8. \( G \leftarrow \text{diag}(x(t)) \)
9. \( a \leftarrow \lambda R^{-T} g_p \)
10. \( u \leftarrow 0 \)
11. \( \alpha_0 \leftarrow 1 \)
12. for \( i \leftarrow p : n_w \) do
13. \( \begin{bmatrix} R & u & \alpha \end{bmatrix} \leftarrow \text{Givens}(R, u, \alpha, a, i) \)
14. end for
15. \( v \leftarrow v + \frac{\lambda \left( \frac{d(t)}{m} - w_p \right) u}{\alpha_{n_w}} \)
16. end for
17. \( w \leftarrow w - v \)

\[(2.29) \quad \text{and} \quad (2.30) \]

\[
\begin{bmatrix}
z(t, 1) \\
\sqrt{\lambda g_1(t)} s(t)
\end{bmatrix}
= \begin{bmatrix}
z(t, 1) \\
\phi^T(n_w)
\end{bmatrix}
\]

\[
T(t, 1) \quad : \quad T(t, n_w)
\]

\[
\begin{bmatrix}
z(t, n_w - 1) \\
\sqrt{\lambda g_{n_w}(t)} s(t)
\end{bmatrix}
= \begin{bmatrix}
z(t, n_w) \\
\phi^T(n_w)
\end{bmatrix}
\]

It is now clear that the derivation does not need to continue, as the regularizing equations \((2.27) - (2.30)\) are in the same form as \((2.19) \text{ and } (2.20)\). Thus, the IQR-RLS update algorithm given in [43] for \(x(t)\) and \(d(t)\) can be performed first as is usual in Algorithm 3. Then the algorithm should be performed \(n_w\) more times, with \(x(t)\) replaced with \(g_1(t)\) to \(g_{n_w}(t)\) and \(d(t)\) replaced with \(g_1(t)s(t)\) to \(g_{n_w}(t)s(t)\). The algorithm for the regularized IQR-RLS-CMAC is shown in Algorithm 5.
2.5.1 Optimizations

Running the IQR-RLS algorithm $n_w$ times for each row would slow the entire algorithm down significantly. However, an important observation to make is that only $m$ rows of $G(t)$ will not be the zero vector. The zero vector rows can be ignored as they would produce a zero $a(t)$ vector. Thus, instead of running the algorithm $n_w$ times more for regularization, it only needs to be run $m$ more times. This optimization is reflected in the for loop in line 6 in Algorithm 5.

Another optimization performed in Algorithm 5 is related to line 7. Here realize that the Givens rotation loop can be started from address $p$ of the $j$'th '1' in the current association vector. This is because $g_p(t)$ is essentially the association vector with every entry, other than the $p$'th entry masked as zero. Therefore every $a_j(t)$ value before the $p$'th address will be zero, making performing the Givens algorithm redundant, as was explained for a similar scenario in Section 2.3.1.

2.6 Conclusions

This chapter presented the IQR-RLS-CMAC which improved the RLS-CMACs computational speed, and also showed a method for implementing regularization/credit assignment into the IQR-RLS-CMAC. Although the IQR-RLS-CMAC reduces the number of calculations required when compared to the RLS-CMAC, the computational complexity remains the same, still making the IQR-RLS-CMAC algorithm difficult to implement on a CPU for multidimensional CMACs. This shows that a totally different algorithm is required if the RLS algorithm is to be used in the CMAC in a computationally efficient manner.

This IQR-RLS-CMAC method however, may still find use in parallel hardware implementations due to its naturally parallelizable structure.
2. INVERSE QR DECOMPOSITION RECURSIVE LEAST SQUARES
CMAC
Kernel Recursive Least Squares CMAC

3.1 Introduction

This chapter introduces the kernel recursive least squares (KRLS) algorithm [92], and shows how to apply it to the CMAC neural network in order to create what we term the kernel recursive least squares CMAC (KRLS-CMAC). The KRLS algorithm provides a means to get around the computational complexity problem found in the RLS-CMAC by significantly reducing the size of the weight vector.

The basic proposal is that the kernel idea from the KLMS-CMAC can be applied to the RLS-CMAC in order to transform the computational complexity from $O(n_w^2)$ to $O(n_w^2)$. Where $n_\alpha$ is the size of a dictionary that is sparsely populated with a subset of the training set, where $n_\alpha << n_w$. The value $n_\alpha$ is also the length of the kernelized weight vector.

Although the KRLS algorithm can use any suitable kernel, here we concern ourselves only with the CMAC kernel, as we wish to create a CMAC that will act in a similar but improved way to the standard CMAC. Also, using the CMAC kernel in the KRLS algorithm allows various CMAC improvements that have been well tested to be applied.

The KRLS and KLMS algorithm can be thought of as a type of growing radial basis function network.

3.2 Introduction to the Kernel Recursive Least Squares Algorithm

The KRLS algorithm was first introduced in [92] and is an online 'kernelized' version of the classic RLS algorithm. In this sense it transforms the RLS algorithm from a linear
3. KERNEL RECURSIVE LEAST SQUARES CMAC

regression solver into a non-linear regression solver. This is also essentially what the RLS-CMAC algorithm does. In the KRLS algorithm, any suitable kernel can be used, where the most common is some sort of Gaussian kernel. A list of potential kernel functions can be found at [93].

The KRLS algorithm works in a similar manner to the KLMS-CMAC, where input vectors are added to a dictionary, and the kernel vector is calculated as a function of the dictionary and current input vector. The main difference is in the learning algorithm used, and the fact that the KRLS algorithm has an inherently built in sparsification method that allows it to be used efficiently in online applications.

3.2.1 Brief Introduction to Kernel Methods

Kernel methods work by non-linearly mapping data to a very high or even infinite dimensional space. The idea is that in a high dimensional space, it will be easy to perform linear regression, or find a linear separator for classification data. The linear fit in the high dimensional space will map back down to a complex curve in the lower dimensional problem space. As the mapping function can be large or even infinite in dimensionality, it can be difficult or impossible to compute. However, if the output of the learning machine depends on the dot product between the high dimensional vector and weight vector, the calculation of the kernel mapping can be avoided through use of a mathematical tool called the ‘kernel trick’. Recently, the kernel trick has been used in works relating to regression and classification such as in the popular support vector machine [55] and in support vector regression [94]. However these methods are generally offline in nature.

In (3.1) there are two equations shown for computing the output of learning machines dependent on the dot product. The left-most equation shows the output calculated in the feature space, where \( \varphi(q) \) calculates the high dimensional feature vector. The right-most equation shows an equivalent calculation performed in the kernel space where the kernel function is defined by an inner product between two feature vectors which is shown in (3.2). The clever trick is that the feature vector actually does not need to be calculated as it is in (3.2), since the kernel trick can be used to create an equation that calculates \( k(Q, q) \) directly, without the need to calculate the feature vector. Note that here \( q \) refers to the current input vector and \( Q \) refers to the sparse dictionary of previously seen input vectors. Vector \( \alpha \) is the weight vector in the kernel

46
space, whose dimensionality is equivalent to the length of the dictionary.

\[ y = \varphi(q)^\top w = k(Q, q)^\top \alpha \quad (3.1) \]

\[ k(Q, q) = \varphi(Q)^\top \varphi(q) \quad (3.2) \]

The difference with the KLMS-CMAC and the KRLS-CMAC compared to the stand-
dalone KRLS algorithm is that the kernel trick is not used, thus the feature vector is
still calculated, and \( k(Q, q) \) is calculated using (3.2). The feature vector in CMAC
terminology is known as the association vector. The association vector is high dimen-
sional, but it is also sparse, and thus the inner product calculation is computationally
efficient, if the number of layers \( m \) is low. Using this method allows certain CMAC
improvements that work by modifying the association vector to be applied, which is
discussed in the later chapters.

In Chapter 7, a continuous KRLS-CMAC that directly calculates the kernel vector
by using the kernel trick, and thus does not calculate the association vector is derived.
It also attempts to apply some of the CMAC improvements, but becomes much more
computationally complex in the process.

### 3.2.1.1 General Kernel Example

In Fig. 3.1, a visualization of how a kernel can be used to distinguish two different
classes in a classification problem is shown. In Fig. 3.1a some two dimensional data
is shown, where the solid circles indicate the first class and the hollow circles indicate
the second class. It is impossible to linearly separate this data as it is. Fig. 3.1b
shows the data transformed by a polynomial kernel given by \( x^2 + y^2 \) (where \( x \)
and \( y \) are the coordinates of the data), which non-linearly transforms the data into three
dimensional space. Now in Fig. 3.1c it is possible to distinguish the two classes in the
three dimensional space using a linear hyperplane. Finally, in Fig. 3.1d it can be seen
that the linear hyperplane across the kernel function is equivalent to a circle in the two
dimensional problem space. Thus, this classification problem was solved with a simple
linear hyperplane, rather than the more complex circle.

This was a simple example. For more complex problems with complex decision
boundaries higher dimensional kernels may be used.
3. KERNEL RECURSIVE LEAST SQUARES CMAC

![Figure 3.1: Kernel Classification Illustration](a) Linear separation of the two classes is impossible. (b) Apply a polynomial kernel to non-linearly transform the data to a higher dimension. (c) Fit a linear hyperplane to the data. (d) Equivalent decision boundary back in the problem dimension.

3.3 KRLS-CMAC Algorithm

The KRLS-CMAC was originally described in the author's previous work in [44], and here is shown in Algorithm 6. In essence, the basic KRLS-CMAC is the just KRLS algorithm, where the kernel vector is generated by (1.11), which uses the CMAC association vector as the feature vector.

3.3.1 CMAC Kernel Advantage

The advantage of using the CMAC association vector to generate the kernel vector is that CMAC improvements that have been built up over the years are more easily applied than if a more standard kernel vector generation algorithm is used. For example learning interference still exists when the Gaussian kernel function is used. If the CMAC feature vector from (1.11) is used, regularization/credit assignment can easily be applied. Similarly eligibility is easily applied if the CMAC association vector is used. Higher order sensitivity functions can be also be implemented to reduce the discrete and non-smooth nature of the binary CMAC kernel, and this is discussed in Chapter 4.
Algorithm 6 KRLS-CMAC Algorithm

1: procedure KRLS-CMAC(i, d)
2:   \( K^{-1} \leftarrow \frac{1}{m} \), \( \alpha \leftarrow \frac{d_1}{m} \), \( P \leftarrow 1 \)
3:   for \( t \leftarrow 2, 3 \ldots n_t \) do
4:     \( q \leftarrow \text{Quantize}(i_t) \)
5:     \( x \leftarrow \varphi(q) \)
6:     \( k \leftarrow Xx \)
7:     \( y \leftarrow k^\top \alpha \)
8:     \( k \leftarrow x^\top x \)
9:     \( a \leftarrow K^{-1}k \)
10: \( \delta \leftarrow k - k^\top a \)
11: \( e \leftarrow d_t - y \)
12: if \( \delta > \nu \) then
13:     \( X \leftarrow \begin{bmatrix} X \\ x^\top \end{bmatrix} \)
14:     \( K^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta K^{-1} + aa^\top & -a^\top \\ -a & 1 \end{bmatrix} \)
15:     \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \alpha - ae \\ e \end{bmatrix} \)
16:     \( P \leftarrow \begin{bmatrix} P & 0 \\ 0 & 1 \end{bmatrix} \)
17: else
18:     \( \nu \leftarrow \frac{Pa}{1 + a^\top Pa} \)
19:     \( P \leftarrow P - \nu a^\top P \)
20: \( \alpha \leftarrow \alpha - K^{-1} \nu e \)
21: end if
22: end for
23: return \( y \)
24: end procedure

3.3.2 Online Sparsification

Sparsification is a necessary aspect of the KRLS algorithm. Without it, every training instance presented to the algorithm would be stored in the dictionary, and the memory use and computational time would grow without bound. Sparsification permits only a useful subset of the training set to be stored in memory and used to calculate the solution. Additionally, sparsification helps to avoid the problem of over training by effectively regularizing the solution, and thus also improving the generalization ability.
3. KERNEL RECURSIVE LEAST SQUARES CMAC

Central to the KRLS algorithm is its online sparsification technique that sparsifies by preventing feature vectors that are approximately linearly dependent on the dictionary from being added to the dictionary. The full concept and derivation of this sparsification technique will not be repeated here, but can be found in [92]. This sparsification technique can be used because it is likely that the dimensionality of the manifold spanned by the training feature vector samples will be much smaller than the actual feature space dimensionality. If an input vector is linearly dependent on the dictionary, then a linear combination of the already added dictionary input vectors is able to represent the input vector. Thus, its training information can be spread amongst the already existing dictionary values. Using approximate linear dependence brings a certain amount of error into the system, but it is usually small and easily controlled by varying a linear dependence threshold, which is defined as parameter $\nu$.

During the sparsification test a sparsification parameter $\delta$ is calculated which is a measure of how approximately linearly dependent the current association vector input is on the dictionary. If $\delta$ is greater than the approximate linear dependence threshold $\nu$, the association vector $\mathbf{x}$ was not approximately linearly dependent on the dictionary, and will thus be added to the dictionary. The kernel weight vector $\mathbf{a}$ will then grow by increasing the vector size by one and setting the new weight. Also, $\mathbf{K}^{-1}$ and $\mathbf{P}$ will be updated, where $\mathbf{K}^{-1}$ is the inverse of a matrix of kernel vectors corresponding to dictionary values, and $\mathbf{P}$ is a matrix used to calculate the update when a training sample is not added to the dictionary. If the threshold is not met, the weights will still be updated, but a new dictionary value will not be added. Instead the training error will be distributed amongst the already existing weights in the weight vector.

The elements of vector $\mathbf{a}$ represent a weighting on how linearly dependent a particular vector in the dictionary is to the current feature vector. If the current feature vector is already in the dictionary, the entries of vector $\mathbf{a}$ will be all zero except for a single unity entry at the index of the matching dictionary point.

The maximum value of $\delta$ is $m$, and thus the sparsification threshold, $\nu$ should be set to some percentage of $m$, and tuned through trial and error.

Other sparsification techniques also exist such as the min-max method used in [25], and a sparsification technique that takes into account the variance of the desired values which is used in [95].

Sparsification can also work to improve modeling performance as well, as the sparsification process effectively regularizes the training data, preventing overtraining.
3.3.3 KRLS Algorithm Modifications

In [95] a fixed budget KRLS algorithm is presented. This algorithm uses a fixed size dictionary which prunes itself if it grows over the budget size. However, this increases the computational complexity to $O(n^2_b)$.

Whilst the RLS and KRLS algorithms are able to adapt to time varying functions, they cannot adapt as fast as LMS algorithms, since they are not designed for tracking. In [96, 97] the extended KRLS algorithm is presented which solves this problem. In [98] a sliding window KRLS algorithm is presented which also aims to improve tracking performance in time-varying functions.

3.4 KRLS Approximations

The ‘else’ section of Algorithm 6 allows the KRLS-CMAC to 1) make use of data not added to the dictionary and 2) be weakly adaptable to time varying systems. Unfortunately, the update of matrix $P$ can be rather computationally intensive. In the following section an approximation invented by this author is shown which reduces the computational complexity.

3.4.1 Approximation to Matrix $P$

When profiling the KRLS-CMAC, the majority of computation effort was found to be spent in the update of matrix $P$. In order to decrease the time taken to compute $P$, an approximation can be made that has little effect on modeling accuracy. In the matrix $P$, the diagonal contains values that are often orders of magnitude larger than other values. Because of this, and the way that $P$ is used, it can be concluded that the majority of the information is stored in the diagonal. Thus, by updating only the diagonal part of $P$, significant computational speed increases may be obtained. Also, due to the approximation, only the diagonal is non-zero, so the $P$ matrix may be stored as a sparse matrix, further increasing computational speed. The approximation is shown in (3.3) and replaces the update of $P$ on line 19 in the KRLS-CMAC algorithm.

$$P \leftarrow P - \frac{\text{diag}(Pa \circ Pa)}{1 + a^T \! Pa}$$  \hspace{0.5cm} (3.3)

Here the symbol $\circ$ indicates element wise multiplication, and $\text{diag}(\cdot)$ indicates a zero matrix with the vector in the parenthesis placed along the main diagonal.
3. KERNEL RECURSIVE LEAST SQUARES CMAC

<table>
<thead>
<tr>
<th>Table 3.1: Approximation to P MSE Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>No Weight Update</td>
</tr>
<tr>
<td>Approx. Update</td>
</tr>
<tr>
<td>Full Update</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.2: Approximation to P Computation Time Results (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>No Weight Update</td>
</tr>
<tr>
<td>Approx. Update</td>
</tr>
<tr>
<td>Full Update</td>
</tr>
</tbody>
</table>

3.4.1.1 Effect of the Approximation of P

The approximation of P was expected to increase the error produced by function modeling only slightly. Table 3.1 shows the mean squared error (MSE) computed from modeling a 1D sine wave and a 2D, 3D and 4D sombrero function. Each problem used a large data set of 10,000 training data. The results show that the full calculation of P returned the lowest error as expected, but the approximate calculation returned errors that were very similar to the full calculation, with less than 1% difference. Both updating algorithms performed significantly better than when no update to the weights was done.

The computation time results are shown in Table 3.2. The approximate update was shown to be significantly faster for two and higher dimensional problems and the speed increase is especially significant in the four dimensional problem. High dimensional problems take longer to compute as the sparsification algorithm will need to accept more data into the dictionary in order to successfully model the function.

3.5 Conclusions

In this chapter the kernel concept was introduced and then combined with the RLS-CMAC seen in Chapter 2 in order to create the KRLS-CMAC. Doing so removed the main flaw of RLS-CMAC which was its large computational complexity. The KRLS-CMAC achieves a much smaller computational complexity due to its smaller weight vector. Additionally, some approximations were applied to the KRLS update algorithm to further improve computation times, thus ensuring that it is fast enough for real time
use. The approximations were compared with the exact update algorithms and very little difference in error was found between them.
3. KERNEL RECURSIVE LEAST SQUARES CMAC
4

KRLS-CMAC and Higher Order Sensitivity Functions

4.1 Introduction

This chapter will integrate higher order sensitivity functions, which were discussed in Chapter 1 into the KRLS-CMAC algorithm which was introduced in the previous chapter, and investigate several effects of doing so. In the standard CMAC the sensitivity function used produces a value that is either '1' or '0', and is thus called the binary sensitivity function. This binary sensitivity function produces a very undesirable quantized staircase like output which is most noticeable at low resolutions.

The binary basis function can also be viewed as being equivalent to a zero order b-spline. From this equivalence, it then follows that higher order b-splines can also be used as the sensitivity function. By doing so, smoother continuous CMAC outputs can be obtained as higher order sensitivity functions are continuous and not discrete. Higher order sensitivity functions also provide a weak regularizing effect, helping to prevent over training by forcing the output to be smooth. Furthermore, when higher order b-splines are used, high quantizing resolution in the CMAC becomes less important since the input becomes continuous, allowing use of a smaller generalization parameter which improves upon computation speed.

In this chapter it is also shown that the generalization parameter plays a large role in determining the modeling error that the CMAC will produce when using the uniform or diagonal overlay. The full overlay produces a kernel response that is perfect, where as the uniform overlay is simply an approximation to the full overlay, where the generalization parameter plays a part in determining how good the approximation is.
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

An experiment to determine how the generalization parameter for the uniform overlay affects this approximation of the full overlay for various dimensions is performed.

Also in this chapter two multi-dimensional kernel combination functions are investigated. The standard CMAC with a linear sensitivity function uses a radial (multiplication) based combination function, however in [9] it was suggested that a 'minimum based function' should be used instead to improve modeling results. This is suggested because when higher order sensitivity functions are used, the hypercube edges will trail off in response magnitude quickly. This trail off can be problematic in the CMAC because if a reduced number of hypercubes is used, such as with the uniform or diagonal overlay, then the overlays will be unable to represent the entire local generalization area evenly. The minimum function does not have such a drastic trail off along the edges on the hypercubes. An experiment is performed to determine if the minimum sensitivity function actually improves the full overlay approximation.

Additionally, the effect of normalizing the association vector with the two-norm is investigated as another means of improving the kernel response approximation.

4.2 Higher Order Sensitivity B-Spline Algorithm

The sensitivity function is the response given by an activated hypercube. The standard CMAC uses a binary sensitivity function that always produces a '1' when a hypercube is activated. The one dimensional kernel response of this binary sensitivity for a quantization level of $r = 13$ is shown in Fig. 4.1a labeled as binary. Notice that the kernel response is discontinuous and stair like. As the resolution increases (Fig. 4.1b), the response approaches a triangle function.

The kernel response is obtained through use of (1.11) by adding a single test association vector to the dictionary, and finding the resulting kernel values (which will be scalar) for a sample of input points surrounding the dictionary value.

The standard CMAC uses the binary sensitivity function only because of its computational simplicity and ease of hardware implementation. In most cases however it is desirable to use a sensitivity function that can utilize real values for greater accuracy and smoother outputs. In [60] Gaussian sensitivity functions are used and in [58] higher order b-splines are used. Here it is decided that higher order b-splines should be implemented as they are the simplest and make intuitive sense as an extension to the zero order b-spline that is used in the standard CMAC. B-splines also have a well
defined support size, which can be useful in terms of computational performance as it limits the number of non-zero values in the kernel vector.

An $o_s$ order b-spline can be used as the sensitivity function directly in place of the binary sensitivity function. The one dimensional kernel responses of a first order and second order b-spline are shown in Fig. 4.1a and Fig. 4.1b labeled as $o_s = 1$ or $o_s = 2$ respectively. Note that an $o_s$ order b-spline sensitivity function will produce an order $o_k$ kernel response b-spline, where $o_k$ is given by (4.1).

$$o_k = 2o_s + 1$$  \hspace{1cm} (4.1)

Notice by looking at Fig. 4.1 that as the order $o_s$ increases, even the low resolution kernel responses become smooth, and similar to their higher resolution equivalents. This is useful, as it allows low quantizing/normalization resolutions to be used and thus also lower generalization parameters, all whilst producing a smooth output. However, later it will be seen that orders above $o_s = 1$ are not useful for multidimensional CMACs.

Note that as the order increases, the effective support of the kernel response decreases, and thus an increase in generalization ratio $\phi$ may be required to compensate. This is investigated further in Section 4.3.1.

4.2.1 B-Spline Calculation Algorithm

In [58] a recursive calculation of the b-spline sensitivities for the association vector is used. In this work Algorithm 7 is used instead, which simply directly calculates the sensitivity values for an $o_s$ order b-spline. This algorithm is a modified version of the
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

Algorithm 7 Get an $o_s$ order b-spline sensitivity response

1: procedure SENSITIVITYRESPONSE($n, D, h, o_s$)
2:   for $i ← 1, 2 \ldots m$ do
3:     for $j ← 1, 2 \ldots n_d$ do
4:        $v ← n_j + D_{i,j}$ mod $h$
5:        $v ← \frac{1}{o_s!} \sum_{k=0}^{o_s+1} \frac{(o_s + 1)^k}{k!} (-1)^k \left(v + \frac{o_s + 1}{2} - k\right)^{o_s}$
6:     if $j = 1$ then
7:        $s_i ← v
8:     else
9:        $s_i ← s_i v$  \hspace{1cm} \triangleright (OPTION 1 - RADIAL)
10:       $s_i ← \min(s_i, v)$ \hspace{1cm} \triangleright (OPTION 2 - MINIMUM)
11:    end if
12:   end for
13: end for
14: return $s$
15: end procedure

explicit b-spline expression found in [99], and the linear sensitivity function algorithm in [9]. The addition of line 5 and the modulus in line 4 is a modification added in this work that allows the b-spline support size to be adjusted, which corresponds to altering the generalization parameter in the standard CMAC. In Algorithm 7, the variable $n$ is the normalized input vector, which is calculated in the same way as the quantized input mapping is calculated in (1.1), but without taking the floor function as shown in (4.2).

$$n ← r \frac{i - i_{\min}}{i_{\max} - i_{\min}}$$ \hspace{1cm} (4.2)

Note that $x^n_+$ from Algorithm 7 is known as the truncated power function whose definition is shown in (4.3).

$$x^n_+ = \begin{cases} x^n & x > 0 \\ 0 & \text{otherwise} \end{cases}$$ \hspace{1cm} (4.3)

When the b-spline order is $o_s = 1$, the sensitivity function can be referred to as the linear sensitivity function [9], since then the b-spline equation simply describes a straight line. When using the linear sensitivity function, line 6 of Algorithm 7 can be replaced with the triangle function shown in (4.4), which is a simpler function with much improved
4.3 Choosing the Generalization Parameter and Overlay Type

A good kernel function is defined by being positive semi-definite and symmetric. The CMAC uniform overlay based kernel function is an approximation of the perfectly symmetric kernel function that is produced when the full overlay is used. In this work only the uniform overlay is focused upon, as it is widely known [45] that it is much better than the diagonal overlay.

The approximation by the uniform or diagonal overlay is heavily distorted when a small generalization parameter and low quantizing resolution is used. The approximation is further distorted in multidimensional spaces when anything less than the full overlay is used, since the receptive field coverage will be much less. A visual example of top down contours of the kernel responses is shown in Fig. 4.2, where on the left most column labeled order = 0, the binary kernel response becomes more and more distorted as the generalization parameter is reduced. The uniform and diagonal overlays have a reduced number of hypercubes available to put together a smooth symmetric kernel in order to save on memory in the non-kernelized CMAC, and to save on computation complexity in the kernelized CMAC.

Using higher order b-spline sensitivity functions can improve the kernel approximation by smoothing out the roughness of the approximation, as can be seen in Fig. 4.2 under the columns order = 1 and order = 2. Again notice that as the generalization parameter increases, the approximation better matches the ideal full overlay case when \( h = \infty \). However, with the higher order sensitivity functions, even \( h = 10 \) is a decent approximation to the full overlay compared to the binary sensitivity function.

The linear sensitivity function approximates a smooth and symmetric radial kernel function much better than the binary sensitivity function for the same generalization parameter. Also, the binary sensitivity function does not result in a radial kernel response, instead producing a more square response. The work in [100, 101] suggests radial functions may be the best available kernels for general approximators.
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

As it is the generalization ratio \( \phi \) that determines the generalization ability of the CMAC, there are multiple possible combinations of \( h \) and \( r \) that can produce the same desired generalization ratio. Generally, with the KRLS-CMAC, and especially later with the credit assigned KRLS-CMAC, it is a good idea to keep \( h \) as small as possible. A large value of \( h \) will increase computation times due to the increased number of non-zero values in the association vector, and the need for a larger inverse matrix calculation in the kernelized credit assigned CMAC as will be seen later in Chapter 5. However, when using a reduced number of hypercubes such as with the uniform overlay, it is desirable to use a large \( h \), since larger values of \( h \) are able to approximate the full overlay more accurately. The CMAC literature does not yet show a way to choose the value of \( h \) while balancing this trade-off, and this is investigated in the experiments performed in Section 4.4.

4.3.1 Local Generalization Area Sizes with Higher Order Sensitivity Functions

Higher order b-spline sensitivity functions activate a smaller local generalization area for the same generalization ratio \( \phi \) due to their reduced support size, as can be seen in Fig. 4.2. This difference in sizes can make the comparisons between sensitivity functions in the experimental work performed in this thesis difficult. This is because the effective local generalization sizes must be kept constant across the tested sensitivity functions in order to give a fair comparison.

An experiment that compared the kernel responses for a binary sensitivity function and linear sensitivity function was conducted. In this test, the difference between the binary kernel and linear kernel response was recorded for various values of \( \phi_{\text{BINARY}} \) and \( \phi_{\text{LINEAR}} \). For each value of \( \phi_{\text{BINARY}} \) tested, the \( \phi_{\text{LINEAR}} \) value that gave the lowest total absolute difference error between kernel responses was recorded. The recorded values were then plotted, and it was discovered that a simple linear function could describe the mapping between the different sensitivity function kernel response support sizes.

For the linear sensitivity function, an equation given by (4.5) was calculated from the results. This equation allows easy calculation of an equivalent value of the generalization ratio \( \phi \) for the linear sensitivity function, when a binary sensitivity ratio is given.

\[
\phi_{\text{LINEAR}} = 1.3939\phi_{\text{BINARY}} \tag{4.5}
\]
Figure 4.2: Kernel Contours - This figure shows the contours of uniform overlay kernel responses for various generalization parameters and b-spline orders for a two dimensional CMAC. Note that the case where \( r = \infty \) and \( h = \infty \) uses the full overlay and kernel generation algorithm shown in Chapter 7.
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

The minimum based linear sensitivity function altered the size of the kernel response differently to the radial sensitivity function. Its binary equivalent conversion is given by (4.6).

\[ \phi_{\text{LINEAR(MIN)}} = 1.30\phi_{\text{BINARY}} \]  
(4.6)

The same experiment was repeated with the binary sensitivity function and second order b-spline sensitivity function. The results produced the equation shown in (4.7).

\[ \phi_{\text{ORDER2}} = 1.7308\phi_{\text{BINARY}} \]  
(4.7)

Additionally, the Gaussian kernel function was also compared as it is used in the self organizing CMAC (SOCM), which is used in comparison tests in the results chapter. It has a conversion equation given by (4.8).

\[ \phi_{\text{GAUSSIAN}} = 0.5971\phi_{\text{BINARY}} \]  
(4.8)

4.3.2 Combination Functions for the Uniform Overlay and Normalization

The authors of [9] state that they found a minimum based multidimensional linear sensitivity function rather than a radial based linear sensitivity function to perform better for CMACs greater than two dimensions in size if the linear or higher order sensitivity functions were used. To get the minimum based sensitivity function the multiplication based equation in option one shown in line 10 of Algorithm 7 should be changed to option two shown in line 11.

In [100] and [101] evidence supporting radial basis functions for general approximation is given. The authors of [9] however believe that using a minimum based function may be a better choice for the CMAC specifically, as there is a concern that when higher order sensitivity functions are introduced the corners of the hypercube will taper off in magnitude too quickly. This can be a problem in the CMAC due to the low number of receptive fields. The effect is that the local generalization area may not be fully and evenly represented. These claims were not supported in the literature apart from anecdotal evidence given in [9]. In Section 4.4 empirical experiments that determine if this claim is true are performed.

For illustration, a single two dimensional CMAC hypercube generated using option one and option two is shown in Fig. 4.3. As can be seen, the receptive field using the minimum function has a higher magnitude response at the corners, where as the
4.3 Choosing the Generalization Parameter and Overlay Type

![Figure 4.3: Hypercube Topology - Comparison between the topology of multiplication and minimum based 2D hypercubes. Darker colors indicate a larger sensitivity response.](image)

![Figure 4.4: Kernel Response Comparison - Contour comparison between the (a) multiplication based and (b) minimum based sensitivity functions.](image)

multiplication radial based function has almost a zero response at the corners. The corresponding perfect full overlay kernel responses for a two dimensional CMAC are shown in Fig. 4.4. Notice that the minimum function produces a kernel function with squared edges, which according to some works may not be the optimal general approximation kernel [100, 101].

### 4.3.3 Association Vector Normalization

Another way to reduce the difference between the uniform and full overlay kernel responses other than increasing the value of the generalization parameter \( h \), is to normalize the association vector by the two-norm as shown in (4.9).

\[
x = \frac{x}{\sqrt{\|x\|}}
\]  

(4.9)
The reason normalization reduces the difference between the full overlay and uniform overlay is because it forces the uniform overlay's peak kernel value (given by $k = \mathbf{x}^\top \mathbf{x}$) to be constant. When the number of hypercubes $m$ is large, such as with the full overlay, the association vector two-norm will be constant for any input vector that was used to produce that association vector. However, for a low value of $m$ such as with the uniform overlay, the two-norm is not constant and can vary significantly. The two-norm is not constant with the uniform overlay because with a reduced number of available hypercubes there is less averaging of the peak value.

A varying kernel peak value can cause a modeling performance reduction as it adds a sort of noise to the kernel response. Additionally, it can cause further trouble in the KRLS sparsification procedure since the value of $k$ will be noisy.

Normalization forces the value of the peak kernel response given by $k$ to be unity for all input vectors, thus emulating the behavior of the full overlay.

### 4.3.4 Quantizing Resolution with Higher Order Sensitivity Functions

Note that although the CMAC input used with higher order sensitivity functions is real valued and not quantized, the generation of the hypercube locations is still dependent on the quantized resolution.

The number of feature space weights required by a non-kernelized CMAC can be viewed in the kernelized CMAC as the number of basis functions available for use in generating a kernel response. If the generalization parameter increases whilst the generalization ratio is kept constant, the number of feature space weights will increase. Similarly, if the generalization ratio is increased whilst keeping the generalization parameter constant, the number of feature space weights will decrease. Therefore, the larger the generalization ratio, the larger the generalization parameter may need to be in order to have a sufficient number of basis functions for a good kernel approximation when the uniform overlay is used.

B-spline sensitivity functions greater than the first order linear function may thus be a poor choice, as they have a need to use a larger generalization ratio to have an equivalent support size when compared to a lower order b-spline sensitivity function, therefore requiring a larger generalization parameter to be used. This may explain why the work in [9] found b-spline orders greater than one to work poorly in the CMAC.
Algorithm 8 Association vector calculation with higher order sensitivity and normalization.

1: procedure \( \varphi_{HN}(n, o_s, D, h) \)
2: \[ [x \quad \bar{w}] \leftarrow \varphi([n]) \]
3: \[ s \leftarrow \text{SensitivityResponse}(n, D, h, o_s) \]
4: \[ s \leftarrow \frac{s}{||s||} \]
5: \[ x(\bar{w}) \leftarrow s \]
6: end procedure

4.3.5 Association Vector Algorithm with Higher Order Sensitivity and Normalization

In Algorithm 8, the algorithm for calculating the association vector with higher order sensitivity functions and normalization is shown, and denoted as \( \varphi_{HN} \). Note that the two-norm normalization is applied to the sensitivity vector instead of the association vector, which results in a more computationally efficient and equivalent result to (4.9).

4.4 Experiments

4.4.1 Kernel Response Error Function Calculation

The error function used for evaluating an \( n_d \) dimensional CMAC uniform displacement array’s ability to approximate the full overlay is shown in Algorithm 9, where \( n_k \) is the number of elements in \( k_{\text{UNIFORM}} \) or \( k_{\text{FULL}} \), and where the function GetUniformOverlayKernelResponse is shown in Algorithm 10. The error calculated is the mean squared error between the kernel response for the uniform overlay, and the ideal full overlay kernel response, which is calculated using the continuous kernel method that is described later in Chapter 7. Note that Algorithm 10 can be modified to be the function GetFullOverlayResponse by replacing the kernel calculation in line 10 with the continuous kernel calculation described later in Chapter 7. These algorithms can be also modified for an any dimensional CMAC displacement array by simply modifying the number of nested for loops used to sample around the chosen center.

In Algorithm 9, a sampling of kernel center points for each dimension is taken. This is required as different centers will produce slightly different kernel responses when the uniform overlay is used and \( h \) is small, especially when normalization is not used. The sampling distance for finding the kernel value responses around the selected centers needs to be balanced between good resolution and fast computation time. For the
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

Algorithm 9 Function to compare the difference in kernel responses between a full overlay and a reduced overlay such as the uniform or diagonal overlay.

1: procedure GETOVERLAYAPPROXERROR(kFULL, kUNIFORM)
2: \( \phi \leftarrow 1.3939 \phi \)  \( \triangleright \) Set correct generalization constant for response tested
3: for \( \phi \leftarrow 0.05 : 0.05 : 0.4 \) do
4: \( \text{res} \leftarrow \frac{h}{\phi} \)
5: for \( t_1 \leftarrow -10 : 5 : 10 \) do
6: for \( t_2 \leftarrow -10 : 5 : 10 \) do
7:  
8: for \( t_{nd} \leftarrow -10 : 5 : 10 \) do
9: \( y \leftarrow [t_1 \ t_2 \ \ldots \ t_{nd}] \)
10: \( n \leftarrow \text{Normalize}(y) \)
11: \( X \leftarrow \phi_{HN}(n, o_s, D, h) \)
12: \( k \leftarrow X^\top X \)
13: \( \text{GetFullKernelResponse}(h, n_d, o_s, \phi, X) \)
14: \( k_{\text{FULL}} \leftarrow \frac{k}{k} \)
15: \( \text{GetUniformKernelResponse}(h, n_d, o_s, \phi, X) \)
16: \( T \leftarrow k_{\text{UNIFORM}} - k_{\text{FULL}} \)
17: \( \text{ERROR} \leftarrow \text{ERROR} + \frac{\sum_{i=1}^{n_d} \sum_{j=1}^{n_d} \sum_{p=1}^{n_d} T_{i,j,p} T_{i,j,p}}{n_k} \)
18: end for
19: end for
20: end for
21: return ERROR
22: end procedure

two dimensional, three dimensional and four dimensional kernels, sampling distances of \( \delta = \frac{2h}{10}, \delta = \frac{2h}{10} \) and \( \delta = \frac{2h}{6} \) are used respectively.

The minimum sensitivity function generalization ratios are multiplied by 1.30 and the linear sensitivity generalization ratios are multiplied by 1.3939 in order to make their support sizes equivalent as discussed in Section 4.3.1.

All experiments used a maximum and minimum quantization/normalization of \( i_{\text{min}} = 10 \) and \( i_{\text{max}} = -10 \) respectively. The generalization ratio \( \phi \) is tested over \([0.1 : 0.05 : 0.4]\) and kernel centers between \([-10 : 5 : 10]\) were tested in order to get a fair representation of the approximation over the input space.
Algorithm 10 Function to compute the kernel response.

1: procedure GETUNIFORMKERNELRESPONSE\(h, n_d, o_s, \phi, X\)
2: \(c_{i_1} \leftarrow 1\) \(c_{i_2} \leftarrow 1\) \(\ldots\) \(c_{i_{n_d}} \leftarrow 1\)
3: for \(i_1 \leftarrow n_1 - h : \delta : n_1 + h\) do
4: \hspace{1em} for \(i_2 \leftarrow n_2 - h : \delta : n_2 + h\) do
5: \hspace{2em} \ldots
6: \hspace{2em} for \(i_{n_d} \leftarrow n_{n_d} - h : \delta : n_{n_d} + h\) do
7: \hspace{3em} \(y \leftarrow [i_{1} \ i_{2} \ \ldots \ i_{n_d}]\)
8: \hspace{3em} \(n \leftarrow \text{Normalize}(y)\)
9: \hspace{3em} \(x \leftarrow \varphi_{\text{HN}}(n, o_s, D, h)\)
10: \hspace{3em} \(k_{c_{i_1}, c_{i_2}, \ldots, c_{i_{n_d}}} \leftarrow X^T x\)
11: \hspace{3em} \(c_{i_{n_d}} \leftarrow c_{i_{n_d}} + 1\)
12: \hspace{2em} \end for
13: \hspace{1em} \end for
14: \(c_{i_2} \leftarrow c_{i_2} + 1\)
15: \(\end for\)
16: \(c_{i_3} \leftarrow 1\)
17: \(c_{i_1} \leftarrow c_{i_1} + 1\)
18: \end for
19: return \(k\)
20: \end procedure

4.4.2 Sensitivity Functions Approximation Comparisons

In this section the uniform overlay and various sensitivity functions are tested on their ability to approximate a third order b-spline continuous resolution full overlay for a range of generalization parameter \(h\) values between \(h = 5\) and \(h = 40\). The normalized and non-normalized radial and minimum sensitivity functions are tested.

The direct full overlay calculation shown later in Chapter 7 is used for calculating the ideal continuous infinite quantizing resolution full overlay kernel response. The quantized linear sensitivity function is equivalent to a third order sensitivity function when the continuous algorithm is used.

Note that this experiment assumes that the optimal kernel is radial. Algorithm 9 will be used for this experiment.
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

4.4.2.1 Two Dimensional Kernel Response Results

In Fig. 4.5 the results for the two dimensional CMAC comparisons are shown. Solid lines indicate the radial functions and the dashed lines represent the minimum functions. Black lines indicate normalized functions and gray lines indicate non-normalized. First the non-normalized minimum function and non-normalized radial results are compared. The non-normalized minimum was very slightly the better approximator below \( h = 6 \). However, for all larger generalization parameters the non-normalized radial function was the better approximator.

Similar results were obtained for the normalized solutions except that the normalized solutions performed better than the non-normalized solutions at generalization parameters below \( h = 13 \). Above this value of \( h \), the normalized and non-normalized approximations were almost identical.

As per the advice in [9], the minimum function is not expected to perform any better in the two dimensional CMAC. These results showed that the best approximation to the full overlay for the two dimensional CMAC comes from a normalized radial sensitivity function. From the results it is recommended that a radial sensitivity function with a generalization parameter of at least \( h = 10 \) be used, to ensure a good approximation to the full overlay.

4.4.2.2 Three Dimensional Kernel Response

In Fig. 4.6 the three dimensional CMAC kernel approximation results are shown. In these results the effect of the minimum function is much more obvious. For the non-normalized functions, the minimum function is a significantly better approximator for almost all \( h < 13 \), after which the radial function became the better approximator again.

However, with the normalized results, the normalized radial function was the better approximator when compared to the normalized minimum function over all generalization parameter values. The normalized results were also better than the non-normalized results.

Thus, from these results it is recommended that the normalized radial function be used for any value of \( h \) for the three dimensional CMAC. It can also be determined that a generalization parameter of \( h \geq 13 \) should be used.

68
4.4 Experiments

Figure 4.5: 2D Kernel Approximation Errors - Key: Radial (—); Radial (No Norm) (—); Minimum (— — —); Minimum (No Norm) (— — —).

4.4.2.3 Four Dimensional Kernel Response

In Fig. 4.7 the four dimensional CMAC kernel approximation results are shown. The results show that the non-normalized radial function was a very poor approximation of the full overlay at most values of \( h < 19 \). At \( h = 5, 6, 7 \) the error values were 122, 102, 38 and are not shown on the graph due to scaling. The non-normalized minimum function performed significantly better compared to the non-normalized radial at most values of \( h < 19 \). These and the previous results clearly show that without normalization, the minimum sensitivity function is the better approximator for lower generalization parameters, especially as the CMAC dimensionality increases.

For the normalized association vector, the results show that the normalized radial function is the best approximator out of all the functions tested. A good choice for the generalization parameter appears to be most values above \( h = 15 \).

From these results it is recommended that the normalized radial sensitivity function be used with a generalization parameter of at least \( h = 15 \).
Figure 4.6: 3D Kernel Approximation Errors - Key: Radial (—); Radial (No Norm) (—), Minimum (— — —); Minimum (No Norm) (— — —).

Figure 4.7: 4D Kernel Approximation Errors - Key: Radial (—); Radial (No Norm) (—), Minimum (— — —); Minimum (No Norm) (— — —).
4.4.3 Higher Order Sensitivity Resolution Effect

In this section an experiment on the effect of the resolution parameter on the ability of the uniform overlay kernel response to approximate the full overlay kernel response is performed. The experiment was run on a three dimensional CMAC with the generalization parameter set to values of \( h = 10, 13, 20, 25 \). The generalization ratio \( \phi \) was modified between runs in order to change the quantization/normalization resolution as seen by the generalization ratio calculation in (1.9), but rearranged to calculate resolution \( r = \frac{h}{\phi} \). Therefore, the larger the generalization ratio, the smaller the resolution required to maintain the same generalization ratio.

The results are shown in Fig. 4.8. They show that as the value of \( \phi \) is increased, and thus the resolution is decreased, the approximation becomes poorer.

This experiment shows that the quantizing/normalization resolution does play a part in the kernel response approximation capabilities. However, if a generalization parameter is carefully selected from the kernel approximation graphs in Figures 4.5, 4.6 and 4.7, the effect due to resolution should not be too important.

The approximation error is increased as the generalization ratio is increased since the number of basis functions available to construct the kernel is reduced due to the lower resolution as is discussed in Section 4.3.4. Equation (1.5) in Section 1.2.6 shows how the number of basis functions or non-kernel CMAC weight count is calculated. However, it should be noted that the generalization parameter choice affects the approximation quality much more significantly.

For example, for a three dimensional CMAC with \( h = 10 \) and \( \phi = 0.4 \), the number of basis functions available with the uniform overlay is 194. With \( h = 13 \) and \( \phi = 0.51 \), the number of basis functions available is 193. Although the number of basis functions are very similar, the better approximation to the full overlay is when \( h = 13 \), as can be seen in Fig. 4.8.

4.4.4 Multidimensional Kernel Contour Error Visualization

In the following, the contours of multidimensional kernel responses are shown, in order to provide intuition on the distortion caused in the kernel response as the dimensionality increases.

Figure 4.9 shows the kernel response contours for the two, three and four dimensional kernel responses for three values of \( h \). The normalized linear sensitivity function was used. For the three and four dimensional contours, a two dimensional subspace
was used, which sampled over the first two dimensions and the remaining dimensions were sampled at a fixed zero center. The values of $h$ were chosen so that one value gave the largest approximation error, one value a reasonable approximation error, and finally with a set value of $h = 30$.

When $h = 6$ a particularly bad approximation of the full overlay kernel response occurs for all dimensionalities. This can be seen visually in the contours (a), (d), (g) where the kernel response appears oval and very distorted rather than circular in nature. When compared to the full overlay contour shown in Fig. 4.2 there is a vast difference. In the contours labeled (b), (e), and (h) the response is significantly improved, especially around the centers, where the largest contribution to the CMAC output is made. Since the largest response occurs in the centers, it is more important for the centers to be a good approximation than the edges. Finally, $h = 30$ produces a kernel response that is shown in contours (c), (f), (i). It is a much better approximation to the full overlay kernel response.

Note that as the dimensionality increases, it is apparent that the kernel response deteriorates for the same value of $h$ as can be seen with the change in the $h = 30$ contour over the dimensions, with the four dimensional approximation being the poorest.
Figure 4.9: Changes over Dimensionality - Uniform overlay contours where (a)(d)(g) show the worst contour, (b)(e)(h) show a reasonable contour chosen from the error figures, and (c)(f)(i) show a fixed contour at $h = 30$. 
4. KRLS-CMAC AND HIGHER ORDER SENSITIVITY FUNCTIONS

4.4.5 Conclusions

In this section higher order b-spline sensitivity functions were introduced and multiple experiments were performed to determine the best sensitivity function choices and parameters. It was found that b-splines of order one (linear) improved modeling results significantly, and one of the main reasons for this was that they produced smoother and more symmetric kernel responses that approximated the full overlay kernel response much better. Additionally, the linear kernel was radial in nature, which has been shown by other works [100, 101] to be a good kernel choice for general approximation.

The results also showed that the value of the generalization parameter $h$ plays a major role in the uniform overlay's ability to approximate the full overlay. Larger values of $h$ generally resulted in better approximations. An experiment was performed to determine the optimal values of $h$.

It was found that normalizing the association vector with the two norm improved the uniform overlay kernel response approximation. This was because it was found that association vectors produced by the full overlay had a constant two-norm over various input vectors, whereas the uniform overlay with small $h$ did not. Forcing the two-norm to be constant significantly improved the approximation.

Additionally, the effect of using a minimum versus a multiplication radial based multidimensional combination function based on the advice in [9] was investigated. It was found that the minimum function produced superior results when compared to the multiplication function only when $h$ was low and the CMAC dimensionality was above two. However, when normalization of the association vector was used, the minimum function was no longer superior for low $h$, and the radial function was almost always the better approximator.

In conclusion, the results of this chapter recommend the use of a linear sensitivity function with a radial multidimensional combination function and two-norm normalization applied to the association vector. In Chapter 8 these results are tested on a regression problem, and it will be seen that the findings in this chapter are consistent with the regression modeling results.
5

KRLS-CMAC and Credit Assignment

5.1 Introduction

This chapter shows how the learning interference problem that was described in Chapter 1 can be solved by using a method called credit assignment (CA) or regularization in the KRLS-CMAC. Credit assignment described in [52, 102], and the CMAC regularization method shown in [48] are essentially the same process. Although the paper describing the regularization method is to a large extent the work that this chapter is based on, regularization will be mostly referred to as credit assignment. This helps avoid confusion with other types of regularization which should not be confused with the CMAC regularization in [48]. The more common use of the word regularization in the machine learning literature is when a constraint on the complexity of the output, in order to reduce the effects of over training or variance, is implied. CMAC regularization is purely used for the reduction of the learning interference problem.

In this chapter a derivation of the equations required to implement regularization in the KRLS-CMAC based on the regularization algorithm used in the KLMS-CMAC in [48] is shown. Credit assignment/regularization was chosen over other methods such as weight smoothing and Tikhonov regularization because the work in [48] has been previously successful in implementing regularization into the KLMS-CMAC. Also, weight smoothing appears to be impossible to implement in a kernel based CMAC, because the smoothing information in a kernel CMAC is contained in the kernel, rather than in the weights like it is in a standard CMAC. Tikhonov regularization also appears to be too complex to use in a computationally efficient manner in a kernel CMAC.
5.2 Regularization

In Chapter 1 it was shown that learning interference occurs because certain more frequently activated weights can grow to be larger in value than those less frequently updated, and thus they end up supplying most of the output contribution. This can cause spikes in the output, as certain input vectors may activate only the less frequently updated weights when calculating the output, causing the output in some areas of the input space to be much smaller in magnitude than expected. The effect of learning interference can be reduced if the \( m \) activated weights that are being updated are all forced to be similar in magnitude to one another. Regularization modifies the CMAC optimization function to specify this as an extra constraint.

The standard CMAC optimization function is given by (5.1). It is simply the squared error between the desired and actual CMAC output.

\[
\epsilon(t) = \frac{1}{2} (d(t) - y(t))^2 \tag{5.1}
\]

In [48] the optimization function is modified to (5.2) in order to implement regularization.

\[
\epsilon(t) = \frac{1}{2} (d(t) - y(t))^2 + \frac{\lambda}{2} \sum_{i: x_i(t) = 1} \left( \frac{d(t)}{h} - w_i(t) \right)^2 \tag{5.2}
\]

This modified optimization function adds a secondary term that forces the activated weights to be similar to one another by increasing the error when they are not similar enough. The summation limit \( i : x_i(t) = 1 \) is shorthand for selecting all the indices in the association vector that are equal to one, or in other words all the activated weight indices.

The derivation of the regularized KLMS-CMAC will not be repeated here but can be referred to in [48]. In [48] Lagrange optimization is performed on the modified optimization function, and the resulting derivation reveals that the regularized kernel vector can be calculated using (5.3), where \( \lambda \) is a constant representing the desired regularization strength, \( I \) is an identity matrix and \( C \) is a matrix given by (5.4).

\[
k = X (I + \lambda C)^{-1} x \tag{5.3}
\]

\[
C = \sum_{i=1}^{n_x} \text{diag}(X_{i,:}) \tag{5.4}
\]

76
5.2 Regularization

5.2.1 Regularization and Credit Assignment Equivalence

By simply looking at equation (5.3) the parallels with the credit assignment CMAC (CA-CMAC) [52, 53] can be noticed. The matrix C can be viewed as storing the credibility of each hypercube/weight, since it is essentially counting the number of times that each weight has been activated by the association vectors stored in the dictionary. Thus henceforth, matrix C will be referred to as the credibility matrix.

The inverse of the credibility matrix is taken in (5.3) (with the identity matrix added to ensure the inverse is defined), which corresponds to the credit assignment idea that the weights should be updated in proportion to the inverse of the hypercubes credibility. Since the credibility matrix is a diagonal matrix, finding this inverse is efficient as only the m scalar values that correspond to the activated weights in x need to be inverted.

Using (5.2) to derive (5.3) as is done in [48], is equivalent to the end result obtained with credit assignment which bypassed the derivation and created the credibility matrix and equation (5.3) through intuition alone. Outside of this chapter, the term credit assignment will be used to describe both regularization and credit assignment.

5.2.2 Online KRLS-CMAC Regularization

Unlike the KLMS-CMAC where the credibility matrix is updated every iteration, in the KRLS-CMAC, the credibility matrix must be (permanently) updated only every time a new input point is added to the dictionary. This difference occurs simply because the online KLMS-CMAC does not specify any sparsification algorithm, and thus adds every training sample to the dictionary. The iterative update of the credibility matrix can be easily achieved by adding the current association vector to the credibility matrix after diagonalizing it, as is shown in (5.5).

\[ C_t \leftarrow C_{t-1} + \text{diag}(x) \]  \hspace{1cm} (5.5)

5.2.3 Illustration of Kernel Regularization

The credibility matrix modifies the kernel response such that it effectively 'indents' the response around the value of the quantized input sample that has been added to the dictionary. In Fig. 5.1a a non-regularized kernel response for \( h = 20 \) and all combinations of a one dimensional value of \( Q \) and \( q \) is shown, where here \( Q \) is the
5. KRLS-CMAC AND CREDIT ASSIGNMENT

![Figure 5.1: Kernel Response Credit Assignment Effect - The effect of regularization on the kernel response with \( h = m = 20 \) and \( \lambda = 1 \). For a (a) normal kernel response, and (b) credit assigned kernel response with \( Q = 20 \) added to the dictionary.](image)

The dictionary of all possible quantized inputs, whose values correspond to the vectors in the association vector dictionary \( \mathbf{X} \), such that \( \mathbf{X} \) is a function of \( Q \) as \( \mathbf{X}(Q) \).

The effect of regularization can be seen in Fig. 5.1b. Here the effect of a single input at \( Q = 20 \) is shown when the credibility matrix is updated with the corresponding association vector \( \mathbf{x} \). In this example \( \lambda = 1 \) is used, and so the lowest point at \((20, 20)\) is at half the maximum value of \( m \), since \( (I + \lambda C)^{-1} \) will halve the kernel value when all the activated values of the credibility matrix are set to one for a single dictionary entry. As more values are added to the dictionary the kernel response will continue to become further indented.

5.3 Update Equations Derivation

Regularization makes the KRLS-CMAC more computationally complex by requiring a few new steps to be performed in the algorithm. The first new step is to update the credibility matrix when a new training sample is input into the CMAC by using (5.5).

Next, before testing the approximate linear dependence of an input training sample, the inverse kernel matrix \( \mathbf{K}^{-1} \) must be updated to take into account the new value of \( \mathbf{C} \). This is because the batch method for calculating \( \mathbf{K}^{-1} \) that is given by (5.6) shows that \( \mathbf{K}^{-1} \) is dependent on the value of \( \mathbf{C} \).

\[
\mathbf{K}^{-1} = \left( \mathbf{X}(I + \lambda C)^{-1} \mathbf{X}^T \right)^{-1}
\]

This updated \( \mathbf{K}^{-1} \) is required before sparsification because the KRLS sparsification procedure works by assuming that the current input will pass the sparsification test and be added to the dictionary. This is illustrated more clearly in (5.7) where it is
shown how $k$ and $k$ are actually calculated by appending the current association vector to the dictionary first. Thus, if regularization is used, the credibility matrix must also be updated to include the latest input, and therefore the inverse kernel matrix which is a function of the credibility matrix must be updated.

\[
X \leftarrow \begin{bmatrix} X \\ X^T \end{bmatrix}
\]

\[
\begin{bmatrix} k \\ k \end{bmatrix} = Xx
\]  
(5.7)

To update $K^{-1}$, equation (5.6) could be calculated every iteration. However, calculating (5.6) every iteration would be extremely computationally demanding, as it requires an $n_{\alpha} \times n_{\alpha}$ matrix inversion as well as $n_{\alpha} \times m$ multiplications. A more efficient way to update the $K^{-1}$ matrix can be obtained through an application of the Woodbury matrix identity [103].

The first step is to note that $K^{-1}$ can be equivalently updated via (5.8) where $G$ is given by (5.9).

\[
K^{-1} \leftarrow \left( K + XG X^T \right)^{-1}
\]

\[
G = (I + \lambda C_t)^{-1} - (I + \lambda C_{t-1})^{-1}
\]

(5.8)  
(5.9)

Now, take note of the fact that $G$ is rank $m$, and then write $X$ in terms of its columns as shown in (5.10). Next, see (5.11) which writes $XG X^T$ in terms of columns.

\[
X = \begin{bmatrix} x_1 & x_2 & \ldots & x_{n_w} \end{bmatrix}
\]

(5.10)

\[
XG X^T = \sum_{i=1}^{n_w} G_{ii} x_i x_i^T
\]

(5.11)

The summation can then be simplified as shown in (5.12) where $S$ is the set of $m$ indices that describe where $G_{ii}$ is nonzero.

\[
\sum_{i=1}^{n_w} G_{ii} x_i x_i^T = \sum_{i \in S} G_{ii} x_i x_i^T
\]

(5.12)

Now by collecting all the $x_i$ that correspond to the indices in set $S$, a new matrix $Y$ can be created. Similarly, the values of $G_{ii}$ selected by the indices of $S$ should be collected and placed into a new diagonal matrix $H$. This is so that (5.13) is true. This gives us a much smaller matrix to work with, where $X$ was of dimension $n_{\alpha} \times n_w$, and now $Y$
is of dimension $n_\alpha \times m$.

$$XG\mathbf{X}^\top = \mathbf{YH}^\top$$ (5.13)

Next, equation (5.8) can be rewritten using (5.13) to obtain (5.14) which produces the same result, but with smaller matrices. Then, using the new update equation in (5.14), the Woodbury Matrix Identity can be applied efficiently, as is shown in (5.15) and (5.16).

$$K^{-1} \leftarrow \left( K + \mathbf{YH}^\top \right)^{-1}$$ (5.14)

$$E = K^{-1}\mathbf{Y}(H^{-1} + \mathbf{Y}^\top K^{-1}\mathbf{Y})^{-1}\mathbf{Y}^\top$$ (5.15)

$$K^{-1} \leftarrow K^{-1} - EK^{-1}$$ (5.16)

The computation speed of (5.15) can be improved by using an efficient linear equations solver algorithm, such as the backslash operator in MATLAB to calculate the equation $(H^{-1} + \mathbf{Y}^\top K^{-1}\mathbf{Y})^{-1}\mathbf{Y}^\top$ as $(H^{-1} + \mathbf{Y}^\top K^{-1}\mathbf{Y})\mathbf{Y}^\top$, thus saving on computing an inverse matrix.

The weight vector $\alpha$ must also be updated since according to [92], $\alpha$ is batch calculated using (5.17), and is thus a function of $K^{-1}$ which in turn is a function of the credibility matrix.

$$\alpha = K^{-1}\mathbf{PA}^\top \mathbf{d}$$ (5.17)

Note that in (5.17), matrix $\mathbf{A}$ is the matrix of all approximate linear dependence vectors $\mathbf{a}$, which are calculated from the training samples as part of the approximate linear dependence test. Thus the batch method can be extremely slow, as matrix $\mathbf{A}$ can be very large since it contains the approximate linear dependence vectors for every single training sample seen, even those that have not been added to the dictionary. Therefore it is very important to perform this update as an iterative method. Updating the weight vector iteratively will also be very important later when eligibility is introduced for online training. This is because, if the weight vector was to be recalculated via batch methods every time a value is added to the dictionary, the eligibility data in the weight vector would be lost.

By substituting (5.15) into (5.16) and then that into (5.17), equation (5.18) is obtained which shows how to update $\alpha$ iteratively. Conveniently, it simplifies to (5.19), as matrix $\mathbf{E}$ will have been already calculated as part of the inverse kernel matrix.
update equation in (5.15).
\[
\alpha \leftarrow K^{-1}PA^\top d - K^{-1}Y(H^{-1} + Y^\top K^{-1}Y)^{-1}Y^\top K^{-1}PA^\top d
\] (5.18)
\[
\alpha \leftarrow \alpha - E\alpha
\] (5.19)

If the current input is determined to be approximately linearly dependent on the dictionary, and thus not to be added to the dictionary, the original value of C and therefore also K^{-1} and α must be restored to their pre-update values. The values of k and a will also need to be recalculated using the restored values to be used in the non-adding weight update equations. Additionally, if the training is done by using a desired value (not through FBE or FOX training), the error between the desired value and CMAC output will also need to be recalculated due to the change in kernel vector.

5.4 Regularization/Credit Assignment Algorithm

Two methods for implementing credit assignment in the KRLS-CMAC were developed in this work. The first method is shown in Algorithm 11. In this algorithm, the credibility matrix is used in the sparsification process as calculated in the previous section. This changes the dictionary values that will be added, and requires the credit assignment matrix update for K^{-1} to be performed every iteration.

A secondary approximate method is shown in Algorithm 13. This method requires two different K^{-1} matrices to be stored. One uses the credibility matrix in its calculations and the other does not. The advantage of this method is that the credibility K^{-1} matrix update step only needs to be performed when an input vector is added to the dictionary. In this work, credit assignment method one will be referred to as the method that uses the credibility matrix in the sparsification process, and credit assignment method two will be referred to as the method that does not.

In Algorithm 11 and Algorithm 13 the credit assigned KRLS-CMAC is shown for a system that learns using desired values for credit assignment method one and credit assignment method two respectively. For systems that use an external error, the error calculations may be removed.

On line 6 of Algorithm 11 the credibility matrix is updated. Following on from that on line 7 the credit assignment update of K^{-1} is performed using function CAUpdate shown in Algorithm 12. The matrix E is also saved for use in updating α if it is decided through the sparsification test that the training sample is to be added to the
5. KRLS-CMAC AND CREDIT ASSIGNMENT

dictionary. The new kernel vector calculation is shown in lines 8 and 9. On line 13 the weight vector is updated to take into account the new credibility matrix. The weight vector is updated with credibility information only when adding a training sample to the dictionary as the update would need to be undone if the training sample was not added. The credit assignment update algorithm is shown in Algorithm 12.

On lines 20 and 21 the $K^{-1}$ and $C$ matrices are restored to their original values since in the else statement no update of the dictionary is performed. Additionally, on lines 22 and 23 the kernel and approximate linear dependence vectors are recalculated. They must be recalculated as the previously calculated versions were performed with the assumption that the current training sample would be added to the dictionary. Note that the approximate update of $P$ as discussed in Section 3.4.1 can also be used.

5.4.1 A Note on Sparse Matrices

Ideally, $K^{-1}$ would be stored sparsely. In reality it is faster to store it as a dense matrix. This is because sparse matrices are computationally inefficient at growing in size, and this required in line 16 of the credit assigned KRLS-CMAC algorithm.

Additionally, $K^{-1}$ would require an extra sparsifying operation to remove near zero values which would be slow as $K^{-1}$ easily becomes populated with a large number of near zero values, due to floating point numerical precision errors.

5.4.2 Secondary Credit Assignment Method

The secondary credit assignment method works by maintaining two separate inverse kernel matrices, where one is credit assigned and the other is not. The non-credit assigned matrix is used to calculate the approximate linear dependence value for the sparsification algorithm, whilst the credit assigned matrix is used to calculate the weight vectors. Using this method means that the inverse kernel matrix update algorithm only needs to be performed when a weight is added, as credit assignment plays no part in the sparsification procedure.

The secondary credit assignment algorithm is shown in Algorithm 13. Here a $'\ast'$ as a subscript indicates that that particular variable is used as part of the sparsification procedure and is thus not a function of the credibility matrix.

Since the credit assignment operation occurs now only if a weight is to be added to the dictionary, it is clearly a significant advantage over method one in terms of computational performance for some iterations as the costly $K^{-1}$ update does not need to
Algorithm 11 The Credit Assigned KRLS-CMAC (Method One)

1: procedure CA-KRLS-CMAC(i, d)
2: \( K^{-1} \leftarrow \frac{1}{m} \), \( \alpha \leftarrow \frac{d_i}{m} \), \( P \leftarrow 1 \), \( C \leftarrow \text{diag}(x_1) \), \( X \leftarrow x_1 \)
3: \( \text{for } t \leftarrow 2, 3 \ldots n_t \text{ do} \)
4: \( q \leftarrow \text{Quantize}(i_t) \)
5: \( [x \ w] \leftarrow \varphi_{\text{HN}}(n, o_s, D, h) \)
6: \( C_t \leftarrow C_{t-1} + \text{diag}(x) \)
7: \( [K^{-1} \ E] \leftarrow \text{CAUpdate}(C_t, C_{t-1}, X, K^{-1}) \)
8: \( k \leftarrow X(I + \lambda C_t)^{-1}x \)
9: \( k \leftarrow x^T(I + \lambda C_t)^{-1}x \)
10: \( a \leftarrow K^{-1}k \)
11: \( \delta \leftarrow k - k^T a \)
12: \( \text{if } \delta > \nu \text{ then} \)
13: \( \alpha \leftarrow E\alpha \)
14: \( e \leftarrow d_i - k^T \alpha \)
15: \( X \leftarrow \begin{bmatrix} X \\ x^T \end{bmatrix} \)
16: \( K^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta K^{-1} + aa^T - a^T \\ -a^T \\ 1 \end{bmatrix} \)
17: \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \alpha - ac \\ e \end{bmatrix} \)
18: \( P \leftarrow \begin{bmatrix} P & 0 \\ 0 & 1 \end{bmatrix} \)
19: \( \text{else} \)
20: \( K^{-1} \leftarrow K_{t-1}^{-1} \)
21: \( C_t \leftarrow C_{t-1} \)
22: \( k \leftarrow X(I + \lambda C_t)^{-1}x \)
23: \( a \leftarrow K^{-1}k \)
24: \( e \leftarrow d_i - k^T \alpha \)
25: \( v \leftarrow \frac{1 + a^T Pa}{Pa} \)
26: \( P \leftarrow P - va^T P \)
27: \( \alpha \leftarrow \alpha - K^{-1}vc \)
28: \( \text{end if} \)
29: \( \text{end for} \)
30: \( \text{return } k^T \alpha \)
31: \( \text{end procedure} \)
be performed every iteration. In this algorithm, both $K^{-1}$ and the non credibility $K_{*}^{-1}$ are updated only when a weight is added. However, this will create a computational increase when adding a dictionary value. Method two may be most useful for applications where an initial training step can be performed offline to populate the dictionary, then afterwards secondary training can be performed very fast online.

The difference in modeling with this method is that the same weights that would have been selected with credit assignment turned off are introduced into the dictionary. The output still exhibits significantly reduced learning interference, but the selected dictionary entries are not as optimal as in method one. In the results chapters the method one and method two credit assignment methods are compared on various test problems.

### 5.5 Choosing the Credit Assignment Strength Parameter

In Fig. 5.2 the results of various values of the credit assignment/regularization strength parameter $\lambda$ are shown when training a two dimensional Sombrero Function. Here, 1000 training samples were used, and results were averaged over 100 runs where each run used a different random seed to generate the training samples. A generalization parameter of $h = 10$ was used with a sparsification parameter of $\nu = 0.5$.

The results show that larger values of $\lambda$ decrease the error more than smaller values. Thus, the credit assignment parameter value should be set to an arbitrarily large value. However, it cannot be set too large as very large values could cause numerical floating point errors, since the inverse or linear equations solver used in line 6 in Algorithm 12 could become a near singular matrix calculation.
Algorithm 13 The Credit Assigned KRLS-CMAC (Method Two)

1: procedure CA-KRLS-CMAC(i, d)
2: \( \mathbf{K}^{-1} \leftarrow \frac{\lambda}{m}, \mathbf{K}_*^{-1} \leftarrow \frac{1}{m}, \alpha \leftarrow \frac{d_i}{m} \), \( \mathbf{P} \leftarrow 1, \mathbf{C} \leftarrow \text{diag}(\mathbf{x}_1), \mathbf{X} \leftarrow \mathbf{x}_1 \)
3: for \( t \leftarrow 2, 3 \ldots n_t \) do
4: \( q \leftarrow \text{Quantize}(i_t) \)
5: \( \mathbf{x} \leftarrow \mathbf{\tilde{w}} \leftarrow \varphi_{\text{HN}}(\mathbf{n}, o_s, \mathbf{D}, h) \)
6: \( k \leftarrow \mathbf{Xx} \)
7: \( k \leftarrow \mathbf{x}^\top \mathbf{x} \)
8: \( \mathbf{a}_* \leftarrow \mathbf{K}_*^{-1}k \)
9: \( \delta_* \leftarrow k - k^\top \mathbf{a}_* \)
10: if \( \delta_* > \nu \) then
11: \( \mathbf{C}_t \leftarrow \mathbf{C}_{t-1} + \text{diag}(\mathbf{x}) \)
12: \( \left[ \mathbf{K}_*^{-1} \ E \right] \leftarrow \text{CAUpdate}(\mathbf{C}_t, \mathbf{C}_{t-1}, \mathbf{X}, \mathbf{K}^{-1}) \)
13: \( k \leftarrow \mathbf{X}(\mathbf{I} + \lambda \mathbf{C}_t)^{-1} \mathbf{x} \)
14: \( k \leftarrow \mathbf{x}^\top (\mathbf{I} + \lambda \mathbf{C}_t)^{-1} \mathbf{x} \)
15: \( \mathbf{a} \leftarrow \mathbf{K}_*^{-1}k \)
16: \( \delta \leftarrow k - k^\top \mathbf{a} \)
17: \( \alpha \leftarrow \mathbf{E} \alpha \)
18: \( e \leftarrow \mathbf{d}_t - k^\top \alpha \)
19: \( \mathbf{X} \leftarrow \begin{bmatrix} \mathbf{X} \\ \mathbf{x}^\top \end{bmatrix} \)
20: \( \mathbf{K}_*^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \mathbf{K}_*^{-1} + \mathbf{a} \mathbf{a}^\top - \mathbf{a}^\top \\ -\mathbf{a}^\top \\ 1 \end{bmatrix} \)
21: \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \alpha - \mathbf{ae} \\ e \end{bmatrix} \)
22: \( \mathbf{K}_*^{-1} \leftarrow \frac{1}{\delta_*} \begin{bmatrix} \delta_* \mathbf{K}_*^{-1} + \mathbf{a}_* \mathbf{a}_*^\top - \mathbf{a}_*^\top \\ -\mathbf{a}_*^\top \\ 1 \end{bmatrix} \)
23: \( \mathbf{P} \leftarrow \begin{bmatrix} \mathbf{P} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \)
24: else
25: \( k \leftarrow \mathbf{X}(\mathbf{I} + \lambda \mathbf{C}_t)^{-1} \mathbf{x} \)
26: \( \mathbf{a} \leftarrow \mathbf{K}_*^{-1}k \)
27: \( e \leftarrow \mathbf{d}_t - k^\top \alpha \)
28: \( \mathbf{v} \leftarrow \frac{1}{\mathbf{P} \mathbf{a}} \)
29: \( \mathbf{P} \leftarrow \mathbf{P} - \mathbf{va}^\top \mathbf{P} \)
5. KRLS-CMAC AND CREDIT ASSIGNMENT

\begin{verbatim}
30:     \alpha \leftarrow \alpha - K^{-1}v_e
31:      end if
32:      end for
33:      return k^T\alpha
34:      end procedure
\end{verbatim}

![Figure 5.2: Effect of CA Parameter](image)

**Figure 5.2: Effect of CA Parameter** - The effect of different credit assignment strength parameter values \( \lambda \) on regression error.

### 5.6 Conclusions

In conclusion this chapter took the idea of regularization/credit assignment that had successfully been previously used in the CA-CMAC and KLMS-CMACs, and combined it with the KRLS-CMAC in order to reduce the learning interference problem. This required the derivation of a new update algorithm for the \( K^{-1} \) matrix and weight vector \( \alpha \) which was previously not required in the CA-CMAC or KLMS-CMAC. Additionally, some parallels were drawn between regularization and credit assignment. It was found that they are essentially the same algorithm. A secondary credit assignment algorithm was also developed, which trades off some modeling accuracy for computational speed.

It was shown that for the KRLS-CMAC with credit assignment, using larger values of \( \lambda \) reduces learning interference more. It is recommended that \( \lambda \) be set to an arbitrarily large value, but not so large as to cause floating point precision issues.
KRLS-CMAC and Vector Eligibility

6.1 Introduction

In this chapter vector eligibility is implemented into the KRLS-CMAC for the first time. Eligibility provides a means to improve the online control capabilities of the KRLS-CMAC through an improved learning algorithm that is better at apportioning error correction to the weights. To implement vector eligibility into the KRLS-CMAC, the vector eligibility algorithm that was first used in the FOX-CMAC [6] is studied, and then modified for application in the KRLS-CMAC algorithm.

Also, in this chapter a method to determine the required FOX vector eligibility profile from an impulse response via a genetic algorithm is developed, and a method that shows how non-convergent impulse responses can be dealt with is also described.

6.2 Online Control and Vector Eligibility

In Chapter 1 the FOX-CMAC was reviewed and was shown to be a CMAC implementation that uses vector eligibility to improve the online motion control learning capabilities of the CMAC. In order to control systems with the FOX-CMAC, a connection configuration similar to feedback error learning (FBE) control [30] is used. This connection topology of the FOX-CMAC is shown in Fig. 6.1. Unlike FBE where the output of the feedback controller is used as the CMAC error signal, the FOX-CMAC uses the difference between the state space variable that is actively controlled, and the reference signal as the error signal. The state space variable being controlled may be the output of a position or velocity sensor for example.
6. KRLS-CMAC AND VECTOR ELIGIBILITY

![Diagram of FOX-CMAC and SYSTEM](image)

**Figure 6.1: Connection Example - FOX-CMAC feedback connection where $y_f$ is the feedback controller output, $y_c$ is the CMAC output, $u$ is the system state vector, and $u_2$ indicates the second element of $u$ which is the state space variable being controlled.**

### 6.2.1 Eligibility Profiles

Vector eligibility requires an eligibility profile to be defined which is based on the impulse response of the combined system and feedback controller. Eligibility profiles are introduced in [6] and shown here by (6.1) where $\nabla_t^*$ represents the eligibility profile vector and $c$, $A$, and $b$ are the matrices that describe the combined system and feedback controller in terms of its discrete linearized state space model. The timestep from zero when the weight was last activated and the decay began is represented by symbol $t$.

$$\nabla_t^* = cA^tb$$ (6.1)

Note that for the eligibility profile, vector $c$ is modified to be all zero except for the location that selects the state space variable that is being controlled by the CMAC. This essentially gives the linearized impulse response for the selected state variable.

The linearized discrete state space model can be found if the system model is known. However, for most practical applications, modeling may be too difficult. Instead, the impulse response of the state space variable being controlled can be measured directly, and then the state space matrices can be tuned to match the response as best as possible. In reality it is impossible to apply an impulse, and thus an approximating step or pulse function can be used instead. Basically, it is only the shape of the impulse response that matters as the magnitude simply affects the learning rate of the system, which will need to be fine tuned by hand anyway.

In [6] it is shown that the eligibility profile can be modeled with lower order state space matrices. A very useful class of second order ($\alpha_e = 2$) eligibility profiles is
described in [6] and shown here by the matrices in (6.2), (6.3) and (6.4).

\[
A = \begin{bmatrix}
1 & pt \\
-p_t p_a & 1 - p_t p_b
\end{bmatrix}
\]  

(6.2)

\[
b = \begin{bmatrix}
0 \\
p_t p_a
\end{bmatrix}
\]  

(6.3)

\[c = [p_c, 0]
\]  

(6.4)

By adjusting the values of \(p_t, p_a, p_b\) and \(p_c\) the impulse response can be approximately matched. According to [6], the eligibility profile does not need to exactly match the impulse response, and a fairly rough match is acceptable. Note that a first order \((o_e = 1)\) eligibility profile is equivalent to an exponential decay profile, and a zero order \((o_e = 0)\) profile is equivalent to no eligibility machinery at all.

In [6] it is shown that the CMAC will converge if (6.5) holds true for the eligibility profile approximation. Variable \(\nabla_i\) in (6.5) is the data vector of the impulse response of the state space variable being controlled over time \(T\), and \(\nabla_i^*\) is the modeled eligibility profile response over \(T\). Equation (6.5) essentially says that the sign of the eligibility profile and impulse response must match at a majority of sample points in order to have an eligibility profile that will work well and allow the CMAC to converge.

\[
\sum_{i=1}^{T} \nabla_i \nabla_i^* > 0
\]  

(6.5)

6.3 KRLS-CMAC Eligibility Derivation

In this section a method for using eligibility in the KRLS-CMAC will be derived by examining the FOX-CMAC. First note that, since the KRLS-CMAC with eligibility is similar to the FOX-CMAC in that it uses vector eligibility, it must be connected in the same way that a FOX-CMAC would be connected, which is shown in Fig. 6.1.

6.3.1 Eligibility Update Equation Derivation

To understand how to apply eligibility to the KRLS-CMAC, we must first look at the FOX-CMAC update algorithm shown in (6.6), which is based on the LMS algorithm.

\[
w_{new} \leftarrow w_{old} - \mu e \xi
\]  

(6.6)

In (6.6) \(\mu\) is the learning rate, \(e\) is the system error shown in Fig. 6.1, and \(\xi\) is the eligibility matrix. The eligibility matrix is an \(o_e \times \psi\) matrix that stores the eligibility
6. KRLS-CMAC AND VECTOR ELIGIBILITY

information, where $\psi$ is the decay threshold which is explained shortly. The eligibility matrix is updated by (6.7), which must be performed every iteration.

$$\xi_{i,i} \leftarrow A\xi_{i,i} + b_i x \quad \forall_i$$  \hspace{1cm} (6.7)

By simply looking at equation (6.6), it can be seen that when compared to the standard CMAC update, in place of the typical association vector $x$, is an alternative association vector based on the eligibility matrix $\xi$ and vector $c$ instead. This is defined for clarity as $\bar{x}$ in (6.8).

$$\bar{x} = c\xi$$  \hspace{1cm} (6.8)

Using this new association vector, an eligibility kernel vector, $\bar{k}$ can then be created using $\bar{x}$ in place of $x$. This is shown in (6.9).

$$\bar{k} = X\bar{x}$$  \hspace{1cm} (6.9)

Equation (6.9) can then be used in place of the standard kernel vector in the KRLS-CMAC weight update equations which are used when a data value is not added to the dictionary.

When using eligibility there needs to be a way to track how long the decay process has been occurring for each individual weight, as eventually after a certain number of timesteps the eligibility profile will have decayed to a near zero value, and continually updating the decay will be computationally wasteful. In [6] a shift register filled with weights and eligibility values of length $\psi$ was used. Each entry is shifted right every iteration and weights exiting the shift register are fully decayed. However, with a modern mathematical computing language such as MATLAB, it is simpler to maintain a sparse vector containing eligibility values and decay information. This way only values that are non-zero in the eligibility matrix will be used in the calculations.

In this algorithm, a vector defined as the decay vector $\omega$ is used to store the decay information. The decay vector stores the number of iterations/timesteps that a particular weight has been decaying for. Each iteration when the association vector is calculated, the activated weight indices are written to the vector $\bar{w}$. This vector is then used to select the corresponding indices in $\omega$, whose values are then set to one as in (6.10).

$$\omega_{\bar{w}} \leftarrow 1$$  \hspace{1cm} (6.10)

For every subsequent iteration any value in the decay vector that is greater than zero
is incremented using (6.11). In (6.11), $\omega > 0$ indicates a selection of all the indices in $\omega$ whose values are greater than zero. This notation is used for all vectors henceforth.

$$\omega_{\omega>0} \leftarrow \omega_{\omega>0} + 1$$

(6.11)

Equation (6.11) is used because any value in the decay vector which is greater than zero corresponds to an eligibility value which is currently under decay. These can be efficiently selected in a sparse vector.

Eventually, once a value in the decay vector reaches a decay threshold defined by parameter $\psi$, that decay vector value is reset to zero as in (6.12), and its corresponding row in the eligibility decay matrix is also reset to zero as in (6.13).

$$\omega_{\omega>\psi} \leftarrow 0$$

(6.12)

$$\xi_{\omega>\psi} \leftarrow 0$$

(6.13)

The decay threshold $\psi$ should be set to the number of timesteps required by the eligibility decay profile to reach a stable near zero value. In Fig. 6.2 an example eligibility profile is shown for an inverted pendulum system that is oscillatory and non-convergent. The decay vector threshold should be set to a value that is approximately at the point where the eligibility profile has sufficiently decayed to zero. In practice it is acceptable to eyeball this value from the eligibility profile graph. In the example Fig. 6.2, the eligibility profile has almost entirely decayed after about 600 timesteps. In practice, even lower values may still work well as it is the first peak which determines most of the contribution from eligibility.

### 6.3.2 Vector Eligibility Weight Update

When updating the weights when not adding to the dictionary, the eligibility kernel vector in (6.9) should be used in place of the standard kernel vector in order to generate the eligibility effect. The subsequent updates to the $P$ matrix remain the same as before, except that the eligibility kernel vector $\check{K}$ is used in place of the standard kernel vector $k$ for all calculations.

An eligibility gain given by $g_e$ multiplies the error, and is used to tune the strength of the update performed by eligibility. This gain is required as the error provided by feedback error learning is not an exact error, and is only an indicator of the magnitude and direction of the error.
6. KRLS-CMAC AND VECTOR ELIGIBILITY

![Graph](image.png)

**Figure 6.2: Inverted Pendulum** - An example of an eligibility profile found for an impulse response. **Key:** The impulse response of a system. (---); The approximated second order eligibility profile (-----).

### 6.3.3 KRLS Eligibility Kernel Weight Decay Parameter

The matrix $\mathbf{P}$ in the KRLS algorithm reduces the update to each kernel space weight via a decay dependent on the amount the weight has already been updated. This ensures the weights activated most often in the update algorithm will converge and do not continually oscillate around a set point due to overtraining. Note that this should not be confused with credit assignment, as credit assignment reduces the update strength in the feature space weights and does this only for dictionary values. The matrix $\mathbf{P}$ affects any kernel space weights used, and so this decay does not affect the learning interference problem.

In the standard KRLS algorithm, the exact optimal decay is calculated based on the assumption that the exact error is known. However, with online learning, the exact error is not known, only the error direction and a tunable error magnitude is known. Additionally, with eligibility, the value of $c$ will affect the decay time. As the magnitude of $c$ also affects the error magnitude, it is easier to fix the value of $c$ to a value of unity, and add a new parameter $\tau$ to the $\mathbf{P}$ matrix update equation. By altering the value of $\tau$ the decay strength can be controlled.

$$\mathbf{P} \leftarrow \mathbf{P} - \mathbf{v} \mathbf{a}^\top \mathbf{P} \tau$$  \hspace{1cm} (6.14)
Increasing the value of $\tau$ acts to decrease the decay time, whilst decreasing $\tau$ does the opposite. Having a decay time that is too quick (large $\tau$) may cause the reference set point to be unobtainable as there will not be enough training performed for those weights that decayed too quickly. Setting it too slow (small $\tau$) may cause oscillations around the set point due to overtraining. This parameter however, is generally tolerated over a wide range of values. A value of $\tau = \frac{1}{50}$ works well in almost all online motion control learning problems tested in this thesis.

### 6.3.4 Disturbances to the Reference Signal

According to [6] if the system is disturbed by a change in system dynamics or change in reference value, the eligibility values in the eligibility matrix will become invalid. This can be resolved in two ways, either by resetting the eligibility matrix and decay vector to zero, or if possible adding the disturbance or reference value as an extra input to the CMAC.

### 6.4 KRLS Vector Eligibility Algorithm

In Algorithm 14 the KRLS-CMAC with eligibility is presented, and the eligibility update algorithm is shown in Algorithm 15. In this version of the algorithm the eligibility matrix and decay vector is reset if the reference changes, as is discussed in Section 6.3.4. Note that the approximate update of $P$ as is discussed in Section 3.4.1 can also be used.

### 6.5 Replacing versus Accumulating Eligibility

In the eligibility update equation shown in (6.7), the eligibility values accumulate every time a weight is activated. In [104] it is suggested that replacing the eligibility value with the starting value every time the same weight is activated is superior to accumulating the eligibility value for the CMAC. However, in the this author’s experiments with eligibility in the KRLS-CMAC and FOX-CMAC, accumulating eligibility values were found to provide much better results compared to replacing eligibility values.

This may be due to the fact that [104] only used exponentially decaying eligibility values, which begin at the first timestep at a value of unity. Vector eligibility values on the other hand tend to begin at zero at the first timestep. Thus if replacing values are used there may not be enough time between replacements for the value to decay to a large enough value such that it would make an impact on the weights.
Algorithm 14 KRLS-CMAC with Eligibility

1: procedure ELI-KRLS-CMAC(i, d)
2: \[ K^{-1} \leftarrow \frac{1}{m}, \alpha \leftarrow \epsilon, P \leftarrow 1, X \leftarrow x_1 \]
3: for \( t \leftarrow 2, 3 \ldots n_t \) do
4: \( q \leftarrow \text{Quantize}(i_t) \)
5: \[ [x \ \bar{w}] \leftarrow \varphi(q) \]
6: \[ [\xi \ \omega] \leftarrow \text{EliUpdate}(\text{ref}_t, \text{ref}_{t-1}, \xi, \omega, A, b, \bar{w}, x, \psi) \]
7: \( k \leftarrow Xx \)
8: \( y \leftarrow k^\top \alpha \)
9: \( k \leftarrow x^\top x \)
10: \( a \leftarrow K^{-1}k \)
11: \( \delta \leftarrow k - k^\top a \)
12: if \( \delta > \nu \) then
13: \[ X \leftarrow [X \ x^\top] \]
14: \[ K^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta K^{-1} + aa^\top & -a^\top \\ -a & 1 \end{bmatrix} \]
15: \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \alpha - ae_t \\ e_t \end{bmatrix} \)
16: \( P \leftarrow \begin{bmatrix} P & 0 \\ 0 & 1 \end{bmatrix} \)
17: else
18: \( \bar{x} \leftarrow \xi c^\top \)
19: \( \bar{k} \leftarrow X\bar{x} \)
20: \( a \leftarrow K^{-1}\bar{k} \)
21: \( v \leftarrow \frac{Pa}{1 + a^\top Pa} \)
22: \( P \leftarrow P - va^\top P \)
23: \( \alpha \leftarrow \alpha - K^{-1}ve_t g_e \)
24: end if
25: end for
26: return \( y \)
27: end procedure
6.6 Finding the Eligibility Profile

The eligibility profile can be designed to match the impulse response by hand tuning the second order model given in Section 6.2.1. However, complex impulse responses will require higher order eligibility profiles to obtain a better match. Higher order models can be created by making $A$ of dimensionality $o_e \times o_e$, and $b$ and $c$ of length $o_e$, where $o_e$ is the eligibility order desired.

These higher order models can then be made to match the impulse response by using a genetic algorithm to evolve the individual elements of the matrices $A$, $b$ and $c$. A brief introduction and tutorial to genetic algorithms can be found in Appendix B.

The error function of the genetic algorithm can be based on the sum squared error or the total absolute error, where the error is the difference between the evolved eligibility profile and the measured impulse response.

The error function that gave the best results for the genetic algorithm is shown in Algorithm 16. Algorithm 16 minimizes the total absolute error between the impulse response data and the eligibility profile. Furthermore, it also attempts to minimize the number of points at which the signs of the impulse response and eligibility profile differ in order to meet (6.5), by adding a fixed penalty every time a difference occurs. This penalty can be adjusted to fine tune the evolution.

To implement the genetic algorithm, the genetic algorithm implementation in the global optimization toolbox in MATLAB was utilized. Each element in the matrices was evolved, and crossover was allowed only between matching matrices.

---

**Algorithm 15 Eligibility Update Algorithm**

1: procedure $\text{ElIUpdate}(ref_t, ref_{t-1}, \xi, \omega, A, b, \bar{w}, x, \psi)$
2: if $ref_t \neq ref_{t-1}$ then
3:     $\xi \leftarrow 0$
4:     $\omega \leftarrow 0$
5: end if
6: $\omega_\psi \leftarrow 1$
7: $\xi_{\omega>\psi} \leftarrow 0$
8: $\omega_{\omega>\psi} \leftarrow 0$
9: $\xi_{:,i} \leftarrow A\xi_{:,i} + b_i x$ \quad $\forall i$
10: $\omega_{\omega>0} \leftarrow \omega_{\omega>0} + 1$
11: return $[\xi \quad \omega]$
12: end procedure
6. KRLS-CMAC AND VECTOR ELIGIBILITY

Algorithm 16 Genetic Algorithm Error Function for Evolving an Eligibility Profile

1. procedure ELPROFILEERRORFUNC
2. for $i \leftarrow 1, 2 \ldots T$ do
3. $\nabla_i^* \leftarrow cA^tb$
4. error $\leftarrow$ error + $|\nabla_i - \nabla_i^*|$
5. if $\text{sgn} (\nabla_i) \neq \text{sgn} (\nabla_i^*)$ then
6. error $\leftarrow$ error + 0.01
7. end if
8. end for
9. return error
10. end procedure

Note that the MATLAB command 'pem', or 'n4sid' can also be used to create a state space model of a specified order from impulse response data. However, in testing these methods were harder to use as they tended to often generate models that would not converge to zero even after modifying the data using the method explained in the next section, whereas the genetic algorithm was always able to evolve a suitable model.

6.6.1 Non-Convergent Impulse Responses

If the impulse response of the system is oscillatory, non-convergent or divergent, an eligibility profile response that does decay to zero is still required for vector eligibility. In this case, the raw impulse response data cannot be used directly, as the genetic algorithm will evolve an eligibility profile to match the non-convergent impulse response data. To get around this, a method to force the raw impulse response data to converge was created. The steps for evolving an eligibility profile for a non-convergent impulse response are listed below.

**Step 1:** The maximum and minimum peaks of the raw impulse response data are extracted, and their magnitude and timestep values recorded.

**Step 2:** The peak values are forced to decay to zero by multiplying their magnitudes with a value less than unity that decreases exponentially as the timestep increases.

**Step 3:** A shape-preserving interpolant such as the piecewise cubic hermite interpolation (PCHIP) method found in the curve fitting toolbox in MATLAB is used to fit a curve to the forcefully decayed data created in step two. A shape-preserving interpolation method must be used as it maintains the peak positions where other interpolation methods do not.
6.7 Conclusion

**Figure 6.3: Impulse Response with Forced Decay** - An example of how a non-convergent impulse response should be forcefully decayed. **Key:** Forced decay model (—); Impulse response (——).

**Step 4:** A genetic algorithm is used to evolve the $A$, $b$ and $c$ matrix and vectors to match the new modified fitted curve.

This method ensures the timestep positions of the impulse response peaks do not change, which is crucial as the peak positions are much more important than their magnitudes. Simply multiplying the impulse response data by an exponentially decreasing value would move the peak positions, thus changing the shape of the response.

Figure 6.3 shows an example of a non-convergent impulse response that was forced to converge via the aforementioned method. The dashed line shows the non-convergent impulse response, and the solid line shows the forced decay curve. The extracted peaks are indicated by the markers on the solid line curve. Notice that the shape-preserving interpolant keeps the peaks at the same location as the impulse response peaks. To extract the peaks a third party MATLAB function called 'extrema' was used which can be found at the link provided in [105].

6.7 Conclusion

In this chapter vector eligibility was introduced and implemented into the KRLS-CMAC algorithm in order to improve the online learning performance. The derivation of the eligibility based KRLS-CMAC algorithm was based on the idea of vector eligibility, which was first implemented in the FOX-CMAC. The FOX-CMAC algorithm was inspected and found to be sufficiently similar to the standard LMS weight update algorithm, which allowed implementation of vector eligibility in the KRLS-CMAC to be readily achieved. It was also shown how the eligibility algorithm can maintain computational efficiency through a sparse decay vector that determines when an eligibility value has
6. KRLS-CMAC AND VECTOR ELIGIBILITY

decayed to zero. The effect of eligibility on the KRLS algorithms P matrix decay was also investigated, and a new tuning parameter was introduced to control this decay.

Additionally, this chapter showed a method to automatically determine working eligibility profile matrices from an impulse response, through use of a genetic algorithm. When the impulse response was non convergent, it was also shown that the impulse response data could be modified by forcing the data to converge and ensuring the peak positions of the impulse response do not shift.
The Quantized and Continuous KRLS-CMACs

7.1 Introduction to the Quantized Improved KRLS-CMAC

In this section the improvements to the CMAC mentioned in the previous chapters are all implemented into a single algorithm called the Improved KRLS-CMAC (IKRLS-CMAC). These improvements are the kernel RLS training algorithm, uniform overlay, higher order sensitivity functions, association vector normalization, credit assignment and vector eligibility. The chapter title uses the word quantized, as later in the chapter a method for a continuous IKRLS-CMAC which has effectively infinite quantizing resolution is introduced.

Some small improvements to the IKRLS-CMAC that were not significant enough to warrant an entire chapter are also discussed.

7.2 Quantized IKRLS-CMAC Algorithm

The quantized IKRLS-CMAC algorithm is shown in Algorithm 17. The normalization function can be found in equation (4.2), the association vector function \( \varphi_{HN} \) can be found in Algorithm 8, SensitivityResponse can be found in Algorithm 7, EliUpdate in Algorithm 15 and CAUpdate in Algorithm 12. Refer to the glossary at the beginning of this thesis for a complete nomenclature list.

7.3 Abbreviations

To help differentiate between the various configurations of the quantized IKRLS-CMAC that will be used later in the test chapters, various abbreviations will be used for
Algorithm 17 The Improved KRLS-CMAC

1: procedure IKRLS-CMAC

2: \( K^{-1}_1 \leftarrow \frac{1}{m} \), \( \alpha_1 \leftarrow e \), \( P_1 \leftarrow 1 \), \( C_1 \leftarrow \text{diag}(x_1) \), \( X_1 \leftarrow x_1 \)

3: for \( t \leftarrow 2, 3 \ldots n_t \) do

4: \( n \leftarrow \text{Normalize}(i_t) \)

5: \( \begin{bmatrix} x & \bar{w} \end{bmatrix} \leftarrow \varphi_{HN}(n, \alpha, D, h) \)

6: \( \begin{bmatrix} \xi & \omega \end{bmatrix} \leftarrow \text{EliUpdate}(ref_t, ref_{t-1}, \xi, \omega, A, b, c, \bar{w}, x, \psi) \)

7: \( C_t \leftarrow C_{t-1} + \text{diag}(x) \)

8: \( [K^{-1}, E] \leftarrow \text{CAUpdate}(C_t, C_{t-1}, X, K^{-1}_{t-1}) \)

9: \( k \leftarrow X(I + \lambda C_t)^{-1}x \)

10: \( k \leftarrow x^T(I + \lambda C_t)^{-1}x \)

11: \( a \leftarrow K^{-1}k \)

12: \( \delta \leftarrow k - k^T a \)

13: if \( \delta > \nu \) then

14: \( X \leftarrow [X \ x \ a^T] \)

15: \( K^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta K^{-1} + aa^T & -a^T \\ -a & 1 \end{bmatrix} \)

16: \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta E \alpha_1 - a e_{g_1} \\ e_{g_i} \end{bmatrix} \)

17: \( P \leftarrow \begin{bmatrix} P & 0 \\ 0 & 1 \end{bmatrix} \)

18: else

19: \( K^{-1} \leftarrow K_{t-1}^{-1} \)

20: \( C_t \leftarrow C_{t-1} \)

21: \( \bar{x} \leftarrow \xi c^T \)

22: \( \bar{k} \leftarrow X(I + \lambda C_t)^{-1}\bar{x} \)

23: \( v \leftarrow \frac{1 + a^T P \bar{a}}{P \bar{a}} \)

24: \( P \leftarrow P - va^T P \tau \)

25: \( \alpha \leftarrow \alpha - K^{-1}ve_{g_2} \)

26: end if

27: end for

28: end procedure

brevity. An abbreviation will consist of three parts. The first part will describe the
credit assignment method used, the second the sensitivity function used and thirdly if
eligibility is used. If any one of these parts is missing, then that particular improvement
is not used.

For the first part, 'CA' is used to indicate the standard credit assignment method
and 'CA2' is used to indicate the secondary credit assignment method. For the second part 'B' is used to indicate the binary sensitivity, 'L' the linear sensitivity and 'O2' the second order b-spline. Finally for the third part, 'E' is used to indicate if vector eligibility is used.

For example, a quantized CMAC with the KRLS training algorithm, Credit Assignment, Linear sensitivity, and Eligibility will be referred to as a CALE-IKRLS-CMAC, and will generally be further shortened to just CALE-CMAC for brevity. The same CMAC without eligibility would be denoted as the CAL-CMAC. A Binary sensitivity CMAC with Eligibility but no credit assignment would be denoted as a BE-CMAC. A summary of all the abbreviations can be found in the glossary.

7.4 Discussion

Combining the improvements is relatively straight forward, however there are a few things to be aware of in the combined algorithm. The first is that due to the use of higher order sensitivity functions, the association vector no longer simply consists of zeros and ones, which credit assignment relies on to do its credibility counting. Fortunately, credit assignment works just as well with the higher order association vectors and non-integer credibility counting. This can be seen in Fig. 7.1 where an example of the kernel indentation similar to Fig. 5.1 in Chapter 5 is shown, but this time with the linear sensitivity function used. Note that the strength of the credit assignment given by parameter $\lambda$ will change slightly with higher order basis functions, however if the association vector is normalized and $\lambda$ is set large enough the difference is negligible.

Two norm normalization of the sensitivity values is also required if vector eligibility is used, as the power of the association vector affects the learning strength and KRLS weight decay equations. Two norm normalization also ensures the kernel vector is in the same range as the continuous CMAC kernel vector which will be seen later in this chapter.

Also note that when calculating the eligibility kernel vector on line 22 of Algorithm 17 with credit assignment, the kernel vector calculation equation requires use of the credibility matrix as well.
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

![Graphs](image)

**Figure 7.1: Credit Assignment Effect** - The effect of credit assignment on the linear sensitivity kernel response with $h = 20$ when $\lambda = 1$. For (a) normal kernel response, and (b) credit assigned kernel response with $N = 20$ added to the dictionary.

7.5 Minor Improvements to the IKRLS-CMAC

In this section minor improvements that can be applied to the IKRLS-CMAC which were not mentioned in previous chapters are briefly discussed.

7.5.1 Output Derivative Learning

Optionally, a step called output derivative limiting that was first used in [6] can be performed on the weights. Output derivative limiting reduces the maximum derivative magnitude of the CMAC output $y$ by updating the weights a second time. The second error used is the previous timestep CMAC output $y_{t-1}$ multiplied by a small gain $g_d$, where $g_d$ is usually set at around 1-10% of $g_e$.

Limiting the derivative prevents the CMAC from attempting to change control signals too fast, which can be a problem in systems with actuators since they have a maximum slew rate.

7.5.2 Initial Gain

An initial gain parameter $g_i$, that multiplies the error used when adding weights to the dictionary can be used. The more divergent the system and feedback controller is, the higher this should be set. It acts as an immediate restoring force which prevents highly divergent systems diverging before the eligibility training has a chance to make an impact.
7.6 Introduction to the Continuous Improved KRLS-CMAC

In the quantized KRLS-CMACs discussed in the previous section, the kernel vector is a function of the dictionary of stored association vectors and the current association vector. In Chapter 4, it was shown that use of the full overlay is beneficial for modeling, but is too much of a computational burden for use in multi-dimensional KRLS-CMACs due to the large number of layers \( m \) required.

In the following sections, a direct continuous method for calculating the kernel vector without use of the association vector is shown. This method is based on the thesis author's previous work in [106]. This direct method also implicitly utilizes the full overlay. The trade-off from using this method however, is that applying CMAC improvements such as higher-order basis functions, credit assignment and vector eligibility becomes more difficult and/or significantly more computationally intensive. As will be seen, it is possible to combine the continuous KRLS-CMAC with higher order sensitivity functions and vector eligibility in a reasonably efficient manner, but it is not so easy to apply credit assignment efficiently.

The work in the following sections may also help to provide new insights on how the CMAC operates.

7.7 Direct Kernel Vector Calculation

This section shows how the CMAC kernel vector can be calculated directly, which allows the 'kernel trick' to be properly utilized like it is in a kernel machine. This way the association vector does not need to be calculated at all.

7.7.1 Quantized Direct Kernel Vector Calculation

A key realization to note is that in the quantized IKRLS-CMAC (when the binary sensitivity function is used), the individual kernel vector value entries in \( k(X, x) \) represent the number of shared hypercubes between the current quantized input association vector \( x(q) \), and the quantized dictionary association vectors in \( X(Q) \). Where here the dictionary of quantized input vectors is defined as \( Q \). Note that the function notation of vectors and matrices is used here to help show the differences between the quantized and continuous IKRLS-CMACs.

Consider a full overlay two dimensional CMAC with a single dictionary entry at \( Q = [3 \ 3] \), where \( h = 3 \) and thus \( m = 9 \). If an input of \( \overline{q} = [3 \ 3] \) is given, then \( \overline{q} \) is
equal to the dictionary entry, and thus the kernel vector value will be a maximal value of \( k = 9 \), since all \( m = 9 \) hypercubes will have been activated. The bar on \( \bar{q} \) denotes that the input of \( \bar{q} \) is a value within the grid space of 1 to \( 2h - 1 \) in all dimensions. If all \( \bar{q} \) surrounding \( Q \) are calculated, then Fig. 7.2 can be generated, where each number in the grid represents the number of hypercube overlaps (kernel value) and where each dimension represents \( \bar{q}_1 \) and \( \bar{q}_2 \).

The number of overlaps/kernel values in the grid can be generated directly using (7.1). Equation (7.1) works by first assuming that the values of \( \bar{q} \) remain within the grid space of 1 to \( 2h - 1 \). It then multiplies the \( \bar{q} \) entries together if they are within the range 1 to \( h \). If they are not within that range, they are first shifted to an equivalent value in that range through the \( 2h - \bar{q} \) part of the equation before being multiplied.

\[
\text{overlaps} = \prod_{j=1}^{n_d} \min(\bar{q}_j, 2h - \bar{q}_j) \tag{7.1}
\]

In a CMAC, the values of \( q \) will extend far past the maximum \( \bar{q} \) value shown in Fig. 7.1 of \( 2h - 1 \), and also any value of \( Q \) with potentially many dictionary entries may be used. In order to use (7.1) for any value of \( Q \) and \( q \), their values must be translated into an equivalent \( \bar{q} \) vector whose values reside between 1 and \( 2h - 1 \). This can be performed with (7.2), where the max function is used to zero any two vectors that are further than a distance of \( h \) apart, and thus do not share any local generalization area. Henceforth, index \( i \) will be used to represent the dictionary index, and \( j \) the dimension.

\[
\bar{q}_{i,j} = \max(h - |Q_{i,j} - q_j|, 0) \tag{7.2}
\]

Now, by substituting (7.2) into (7.1), equation (7.3) is created. It can be determined that the first term of (7.3) given by \( \max(h - |Q_{i,j} - q_j|, 0) \) will always be the minimum.

\[
k_i = \prod_{j=1}^{n_d} \min(\max(h - |Q_{i,j} - q_j|, 0), \max(h + |Q_{i,j} - q_j|, 0)) \tag{7.3}
\]

Thus, to find the first order b-spline kernel vector for any \( Q \) and \( q \), equation (7.4) can be used.

\[
k_i(Q, q) = \prod_{j=1}^{n_d} \max(h - |Q_{i,j} - q_j|, 0) \tag{7.4}
\]

Using this method to generate the kernel vector means that instead of storing association vectors in the dictionary, the quantized input vector should be stored instead.
7.7 Direct Kernel Vector Calculation

Figure 7.2: Kernel Function Insight - Shows kernel vector values for $Q = [3 \ 3]$, $\bar{q} = [3 \ 3]$ at the center, and all nearby possible $\bar{q}$ within the local generalization area of $h = 3$.

There is also no need to calculate the association vector using this method, since $k(Q, q)$ is now directly a function of the quantized input $q$, rather than previously when it was a function of the association vector like in $k(X(Q), x(q))$.

### 7.7.2 Continuous Direct Kernel Vector Calculation

Conceptually, for the CMAC, it makes sense that only a discrete number of hypercubes should overlap each other. However, it is possible to simply omit the quantization, in (7.4) and allow for a real number of hypercubes to overlap instead. Doing so is equivalent to bringing the number of hypercubes and quantizing resolution to infinity. This can be done by, instead of quantizing the input vector, normalizing with (4.2).

The kernel vector equation then becomes what is shown in (7.5) where $N$ is a dictionary of normalized input vectors.

$$k_i(N, n) = \prod_{j=1}^{n_d} \max (h - |N_{i,j} - n_j|, 0) \quad (7.5)$$

This kernel vector algorithm can be used in the KRLS-CMAC in Chapter 3. All other calculations in the KRLS-CMAC remain the same, except for the kernel vector calculation which is now calculated directly.

### 7.7.3 Higher Order Kernels

Equation (7.5) generates a binary sensitivity CMAC kernel ($a_k = 1$, $a_s = 0$). As shown in Chapter 4, a quantized CMAC with higher order b-splines used as the sensitivity function will cause a kernel response that is a b-spline of an order governed by (4.1).
Algorithm 18 Calculating the kernel vector directly

1: procedure HIghOrderKernel(n, n_\alpha, n_d, o_k)
2:   for i \leftarrow 1, 2, \ldots n_\alpha do
3:     k_i \leftarrow 1
4:     for j \leftarrow 1, 2, \ldots n_d do
5:       v \leftarrow |N_{i,j} - n_j|
6:       v \leftarrow \frac{(o_k + 1)v}{h}
7:       v \leftarrow \frac{1}{o_k!} \sum_{p=0}^{o_k-1} \binom{o_k + 1}{p} (-1)^p \left( v + \frac{o_k + 1}{2} - p \right)
8:       k_i \leftarrow k_i v
9:     end for
10:   end for
11: return k
12: end procedure

With the continuous CMAC, the kernel response is calculated directly, and thus b-splines of order o_k can be used to obtain the same higher order kernel response as in the quantized CMAC.

A modified algorithm that is similar to Algorithm 7 which is used to calculate the sensitivity functions in the quantized IKRLS-CMAC can be used to directly generate a higher order b-spline kernel vector. This algorithm is shown in Algorithm 18.

Algorithm 18 can be potentially much more computationally intensive than Algorithm 7, especially when the third order b-spline kernel is desired. In the continuous CMAC, use of the third order b-spline kernel has a number of operations count given by 27n_\alpha n_d, whereas in the quantized CMAC it is given by 5mn_d + n_\alpha m. By substituting in typical values, it can be determined that generally 27n_\alpha n_d >> 5mn_d + n_\alpha m, assuming the uniform overlay is used and not the full overlay.

Continuous CMACs will be short hand denoted as C_{o_k}-IKRLS-CMAC. Where the numerical value of o_k replaces the symbol. For example, the third order continuous b-spline CMAC is represented as the C3-IKRLS-CMAC, and for brevity when it is clear, just as the C3-CMAC.
7.8 Continuous KRLS-CMAC Credit Assignment

In order to use credit assignment in the continuous KRLS-CMAC, a credit assigned kernel vector needs to be created. In the quantized IKRLS-CMAC, creating a credit assigned kernel vector required the use and manipulation of the association vector. In this section an algorithm for directly generating a credit assigned kernel vector that does not require the use of any association vectors is described.

The credit assignment algorithm for the continuous CMAC was derived by analyzing the credit assignment method used in the KLMS-CMAC and quantized IKRLS-CMAC, whose credit assignment kernel equations are given by (5.3).

7.8.1 Combinatorics Credit Assignment Algorithm

By using a method involving combinatorics calculations, credit assignment can be implemented into the continuous KRLS-CMAC. The same information stored in the credibility matrix can be calculated by determining the number of hypercubes that every combination of dictionary entries shares with both each other, and the current input vector.

The derivation and steps for this algorithm will be presented in the sections below. For simplicity, it will be first assumed that the IKRLS-CMAC is one dimensional only. Later in Section 7.8.2 it will be shown how the algorithm can be generalized to multidimensional IKRLS-CMACs.

An example will be used to help clarify the steps. The example dictionary values are set as \( N_1 = 3, N_2 = 6, N_3 = 10, N_4 = 5, N_5 = 7 \) and the current input is set as \( n = 4 \). These dictionary values are chosen such that they produce the same result in the quantized and continuous KRLS-CMACs, so that comparisons can be drawn. But in a real situation any real valued dictionary values can be used with the continuous method. The resolution is set to \( r = 10 \), and \( m = h = 4 \) as this is a one dimensional example. In this example, at all times the credit assignment strength parameter is set as \( \lambda = 1 \) for simplicity.

First, the quantized method for calculating the credit assigned kernel vector is shown for reference. In (7.6) the corresponding association vectors that would be calculated in the standard kernel vector generation method are shown for the dictionary values
and current input, where $x$ is the current association vector.

\[
\begin{align*}
    x(n) &= [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0]^T \\
    x(N_1) &= [1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0]^T \\
    x(N_2) &= [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]^T \\
    x(N_3) &= [0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1]^T \\
    x(N_4) &= [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]^T \\
    x(N_5) &= [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]^T \\
\end{align*}
\] (7.6)

The original kernel vector for this example calculated with (1.11) is shown in (7.7), where $X(N) = [x(N_1) \ x(N_2) \ x(N_3) \ x(N_4) \ x(N_5)]^T$.

\[
k = X(N)x(n) = [3 \ 2 \ 0 \ 3 \ 1]^T
\] (7.7)

The credibility matrix $C$, after updating it with the current association vector, $x(n)$, as in (5.5) is given by (7.8) (in bold are the entries activated by $x(n)$).

\[
C = \text{diag} \begin{bmatrix} 1 & 4 & 1 & 0 & 4 & 2 & 0 & 3 & 0 & 2 & 3 & 1 \end{bmatrix}
\] (7.8)

And the credit assigned kernel vector, here denoted as $\hat{k}$ for $\lambda = 1$ is given in (7.9) by using (5.3).

\[
\hat{k} = X(N)(I + \lambda C)^{-1}x(n) = [0.7833 \ 0.4 \ 0 \ 0.65 \ 0.2]^T
\] (7.9)

Now the steps for the continuous credit assignment kernel vector algorithm are presented below.

7.8.1.1 Step 1 - Temporarily Append Current Normalized Input to the Dictionary

The first step is to create a temporary dictionary with $n$ appended as the last entry. This step performs the same function as (5.5) which is $C \leftarrow C + \text{diag}(x)$. Additionally, this also allows the value of $k$ to be easily found in a later step.

An example is given in (7.10), where $\hat{N}$ is the temporary dictionary with the current input appended.

\[
\hat{N} = \begin{bmatrix} N \ n \end{bmatrix} = [3 \ 6 \ 10 \ 5 \ 7 \ 4]^T
\] (7.10)
7.8 Continuous KRLS-CMAC Credit Assignment

7.8.1.2 Step 2 - Calculate the Original Kernel Vector

The second step is to calculate the original non-credit assigned kernel vector \( k(\hat{N}, n) \) using (7.5). Note that for multidimensional problems, in this method each dimension is initially calculated individually and not multiplied together.

The example original kernel vector is shown in (7.7).

7.8.1.3 Step 3 - Create \( z \) Vector

A new vector \( z \) is created which is simply a collection of all the non-zero kernel vector values. Note that variable \( n_z \) is used to denote the number of non-zero kernel vector values, and thus also the length of \( z \).

Vector \( z \) is created because any original kernel vector values that are zero will also be zero in the credit assigned kernel vector. By using a smaller vector, significant computational time can be saved in later steps.

For the example vector \( z \) is given (7.11).

\[
z = \begin{bmatrix} 3 & 2 & 3 & 1 & 4 \end{bmatrix} \quad (7.11)
\]

7.8.1.4 Step 4 - Find All Combinations of the Values in \( z \)

Find all combinations of the values in \( z \), and calculate all the \( \gamma \) values. The \( \gamma \) values are the number of hypercubes shared by each combination of the dictionary entries in \( \hat{N} \) that correspond to the \( z \) entries for a particular combination. The example will make this clear. Then, for each combination, take the minimum of the \( z \) and \( \gamma \) values and store the answer.

Next, sort the results by the number of \( z \) combinations. For example, all combinations of two \( z \) entries go into group one, all combinations of three \( z \) entries go into group two and so on. There will be \( n_z - 1 \) groups of combinations in total.

Essentially, what is happening is that the number of shared hypercubes for every dictionary combination involving the current input value is being found.

For the example, all combinations for all groups are calculated and shown in Table 7.1. In Fig. 7.3 an example of three different combinations is shown visually in a CMAC diagram. In the top combination, \( z_1 \) and \( z_2 \) correspond to \( N_1 \) and \( N_2 \). There is one hypercube that shares the dictionary values \( n, N_1, N_2 \). In the middle combination, \( z_1 \) and \( z_3 \) correspond to \( N_1 \) and \( N_3 \). There are two hypercubes that share the dictionary values \( n, N_1, N_3 \). Finally, in the bottom combination \( z_1, z_2 \) and \( z_3 \) correspond to
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

N_1, N_2, N_3. There is only one hypercube that shares dictionary values n, N_1, N_2, N_3.

Taking the combination minimums of z and γ is equivalent to determining the number of hypercubes that a set of dictionary values and the input value share.

For example here it is shown how to calculate \( \min(z_1, z_4, \gamma_3) \) which uses the dictionary input combination N_1, N_5, n. Here \( z_1 = h - |N_1 - n| \), \( z_4 = h - |N_5 - n| \) and \( \gamma_3 = h - |N_1 - N_5| \). Remember that z_4 corresponds to N_5 because k_4 = 0.

Note that, having calculated the minimums in group one, it is possible to easily calculate subsequent groups by noting that each combination in the next highest group can be calculated from three results from the previous group. For example

\[
\min(z_1, z_2, z_3, \gamma_1, \gamma_2, \gamma_4) \text{ is equivalent to } \min(\min(z_1, z_2, \gamma_1), \min(z_1, z_3, \gamma_2), \min(z_2, z_3, \gamma_4)).
\]

To select the three results from the previous group, a pre-calculated look up table can be used. In Appendix C it is shown how this look up table can be generated.
### 7.8 Continuous KRLS-CMAC Credit Assignment

#### Table 7.1: Combination Table for \( n_s = 5, n = 4 \) and \( h = 4 \).

<table>
<thead>
<tr>
<th>Non-Zero Kernel Vector Values (Values of ( z ))</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z_1 )</td>
<td>( 3(N_1 - 3) )</td>
</tr>
<tr>
<td>( z_2 )</td>
<td>( 2(N_2 - 6) )</td>
</tr>
<tr>
<td>( z_3 )</td>
<td>( 3(N_3 - 5) )</td>
</tr>
<tr>
<td>( z_4 )</td>
<td>( 1(N_5 - 7) )</td>
</tr>
<tr>
<td>( z_5 )</td>
<td>( 4(N_6 - 4) )</td>
</tr>
</tbody>
</table>

**Group 1**

- \( \min(x_1, x_2, y_1) \) \( \Rightarrow \) \( \min(3, 2, 1) = 1 \)
- \( \min(x_1, x_3, y_2) \) \( \Rightarrow \) \( \min(3, 3, 2) = 2 \)
- \( \min(x_1, x_4, y_3) \) \( \Rightarrow \) \( \min(3, 1, 0) = 0 \)
- \( \min(x_1, x_5, y_4) \) \( \Rightarrow \) \( \min(3, 4, 3) = 3 \)
- \( \min(x_2, x_3, y_5) \) \( \Rightarrow \) \( \min(2, 3, 3) = 2 \)
- \( \min(x_2, x_4, y_6) \) \( \Rightarrow \) \( \min(2, 1, 3) = 1 \)
- \( \min(x_2, x_5, y_7) \) \( \Rightarrow \) \( \min(2, 4, 2) = 2 \)
- \( \min(x_3, x_4, y_8) \) \( \Rightarrow \) \( \min(3, 1, 2) = 1 \)
- \( \min(x_3, x_5, y_9) \) \( \Rightarrow \) \( \min(3, 4, 3) = 3 \)
- \( \min(x_4, x_5, y_{10}) \) \( \Rightarrow \) \( \min(1, 4, 1) = 1 \)

**Group 2**

- \( \min(x_1, x_2, x_3, y_1, y_2, y_3) \) \( \Rightarrow \) \( \min(1, 2, 2) = 1 \)
- \( \min(x_1, x_2, x_4, y_1, y_3, y_6) \) \( \Rightarrow \) \( \min(1, 0, 1) = 0 \)
- \( \min(x_1, x_2, x_5, y_1, y_4, y_7) \) \( \Rightarrow \) \( \min(1, 3, 2) = 1 \)
- \( \min(x_1, x_3, x_4, y_2, y_3, y_8) \) \( \Rightarrow \) \( \min(2, 0, 1) = 0 \)
- \( \min(x_1, x_3, x_5, y_2, y_4, y_9) \) \( \Rightarrow \) \( \min(2, 3, 3) = 2 \)
- \( \min(x_1, x_4, x_5, y_3, y_4, y_{10}) \) \( \Rightarrow \) \( \min(0, 3, 1) = 0 \)
- \( \min(x_2, x_3, x_4, y_5, y_6, y_8) \) \( \Rightarrow \) \( \min(2, 1, 1) = 1 \)
- \( \min(x_2, x_3, x_5, y_5, y_7, y_9) \) \( \Rightarrow \) \( \min(2, 2, 3) = 2 \)
- \( \min(x_2, x_4, x_5, y_6, y_7, y_{10}) \) \( \Rightarrow \) \( \min(1, 2, 1) = 1 \)
- \( \min(x_3, x_4, x_5, y_5, y_7, y_{10}) \) \( \Rightarrow \) \( \min(1, 3, 1) = 1 \)

**Group 3**

- \( \min(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_5, y_6, y_8) \) \( \Rightarrow \) \( \min(1, 0, 0) = 0 \)
- \( \min(x_1, x_2, x_3, x_5, y_1, y_2, y_4, y_5, y_7, y_9) \) \( \Rightarrow \) \( \min(1, 2, 2) = 1 \)
- \( \min(x_1, x_2, x_4, x_5, y_1, y_3, y_4, y_6, y_7, y_{10}) \) \( \Rightarrow \) \( \min(0, 1, 0) = 0 \)
- \( \min(x_1, x_3, x_4, x_5, y_2, y_3, y_4, y_8, y_9, y_{10}) \) \( \Rightarrow \) \( \min(0, 2, 0) = 0 \)
- \( \min(x_2, x_3, x_4, x_5, y_5, y_7, y_8, y_9, y_{10}) \) \( \Rightarrow \) \( \min(1, 2, 1) = 1 \)

**Group 4**

- \( \min(x_1, x_2, x_3, x_4, x_5, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}) \) \( \Rightarrow \) \( \min(0, 1, 0) = 0 \)
Figure 7.3: Continuous Credit Assignment Example - Shows visually what the taking the minimum of the \( z \) and \( \gamma \) vectors performs for a few example combinations.
7.8 Continuous KRLS-CMAC Credit Assignment

7.8.1.5 Step 5 - Select Combinations

The credit assigned kernel vector is generated one element at a time, from each value of \( z \). To determine \( \hat{k}_1 \), select all combination results calculated in the previous step which involve \( z_1 \). A lookup table is used to select the relevant combination indices, and in Appendix C it is shown how this lookup table can be generated.

Each selected result is denoted with symbol \( \theta_i^j \), where \( j \) is the group selected from, and \( i \) is the index position of the combination result in the selected group.

Table 7.2: Selection Table for \( z_1 \).

<table>
<thead>
<tr>
<th>Non-Zero Kernel Vector Value of Interest</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z_1 )</td>
<td>( 3(N_1 = 3) )</td>
</tr>
<tr>
<td>Group 1</td>
<td></td>
</tr>
<tr>
<td>( \theta_1^1 = \min(z_1, z_2, \gamma_1) )</td>
<td>( \min(3, 2, 1) = 1 )</td>
</tr>
<tr>
<td>( \theta_2^1 = \min(z_1, z_3, \gamma_2) )</td>
<td>( \min(3, 3, 2) = 2 )</td>
</tr>
<tr>
<td>( \theta_3^1 = \min(z_1, z_4, \gamma_3) )</td>
<td>( \min(3, 1, 0) = 0 )</td>
</tr>
<tr>
<td>( \theta_4^1 = \min(z_1, z_5, \gamma_4) )</td>
<td>( \min(3, 4, 3) = 3 )</td>
</tr>
<tr>
<td>Group 2</td>
<td></td>
</tr>
<tr>
<td>( \theta_1^2 = \min(z_1, z_2, z_3, \gamma_1, \gamma_2, \gamma_3) )</td>
<td>( \min(1, 2, 2) = 1 )</td>
</tr>
<tr>
<td>( \theta_2^2 = \min(z_1, z_2, z_4, \gamma_1, \gamma_3, \gamma_6) )</td>
<td>( \min(1, 0, 1) = 0 )</td>
</tr>
<tr>
<td>( \theta_3^2 = \min(z_1, z_2, z_5, \gamma_1, \gamma_4, \gamma_7) )</td>
<td>( \min(1, 3, 2) = 1 )</td>
</tr>
<tr>
<td>( \theta_4^2 = \min(z_1, z_3, z_4, \gamma_2, \gamma_3, \gamma_8) )</td>
<td>( \min(2, 0, 1) = 0 )</td>
</tr>
<tr>
<td>( \theta_5^2 = \min(z_1, z_3, z_5, \gamma_2, \gamma_4, \gamma_9) )</td>
<td>( \min(2, 3, 3) = 2 )</td>
</tr>
<tr>
<td>( \theta_6^2 = \min(z_1, z_4, z_5, \gamma_3, \gamma_4, \gamma_10) )</td>
<td>( \min(0, 3, 1) = 0 )</td>
</tr>
<tr>
<td>Group 3</td>
<td></td>
</tr>
<tr>
<td>( \theta_1^3 = \min(z_1, z_2, z_3, z_4, \gamma_1, \gamma_2, \gamma_3, \gamma_5, \gamma_6, \gamma_8) )</td>
<td>( \min(1, 0, 0) = 0 )</td>
</tr>
<tr>
<td>( \theta_2^3 = \min(z_1, z_2, z_3, z_5, \gamma_1, \gamma_2, \gamma_4, \gamma_5, \gamma_7, \gamma_9) )</td>
<td>( \min(1, 2, 2) = 1 )</td>
</tr>
<tr>
<td>( \theta_3^3 = \min(z_1, z_2, z_4, z_5, \gamma_1, \gamma_3, \gamma_4, \gamma_6, \gamma_7, \gamma_10) )</td>
<td>( \min(0, 1, 0) = 0 )</td>
</tr>
<tr>
<td>( \theta_4^3 = \min(z_1, z_3, z_4, z_5, \gamma_2, \gamma_3, \gamma_4, \gamma_8, \gamma_9, \gamma_10) )</td>
<td>( \min(0, 2, 0) = 0 )</td>
</tr>
<tr>
<td>Group 4</td>
<td></td>
</tr>
<tr>
<td>( \theta_1^4 = \min(z_1, z_2, z_3, z_4, z_5, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9, \gamma_10) )</td>
<td>( \min(0, 1, 0) = 0 )</td>
</tr>
</tbody>
</table>

7.8.1.6 Step 6 - Calculate the Unadjusted Subtraction Vector

When calculating a credit assigned kernel vector in the quantized KRLS-CMAC, several values are selected in the credibility matrix, increased by \( +1 \) because of the identity matrix, inverted and then added together to get the credit assigned kernel vector value.
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

\[
\begin{array}{c|ccccc}
1 & 1 & 1 & 1 & 1 & \frac{1}{2} \\
\hline
1 & 6 & 12 & 20 & 1 & \frac{1}{5} \\
\hline
1 & 1 & 4 & 1 & \frac{1}{3}
\end{array}
\]

Figure 7.4: Tally Visualization - The selected values in the \( C \) matrix can be written in terms of tally counts. Each column contains information on the subtraction contribution.

These selected values in the credibility matrix can be represented visually as tallies. For the example, in Fig. 7.4 the tallies for \( \hat{k}_1 \) are shown, where the values selected by \( x \) in the credibility matrix from (7.8) plus identity matrix are \( (I + \lambda C) \) are \([5 \ 4 \ 3]\).

The subtraction vector is a vector that contains the number of tallies per column, rather than row. To get the subtraction vector, first the unadjusted subtraction vector (denoted as \( \hat{s} \)) must be calculated using (7.12). It is unadjusted because it will contain duplicate shared hypercube counts. (i.e. the earlier entries in the unadjusted subtraction vector are also counting the same hypercubes as the later entries).

\[
\hat{s}_j = \sum_{i=1}^{n_x-1} \theta_i^j
\]  

(7.12)

For the example the unadjusted subtraction vector calculated with (7.12) and Table 7.2 is given in (7.13).

\[
\hat{s} = [6 \ 4 \ 1 \ 0]
\]  

(7.13)

7.8.1.7 Step 7 - Calculate the Subtraction Vector

The subtraction vector is termed as such because its elements are used to subtract from the original kernel vector value to get the credit assigned kernel vector value.

Algorithm 19 is used to adjust the values in the unadjusted subtraction vector, to produce the subtraction vector (denoted as \( s \)). This algorithm removes the duplicate shared hypercube counts, and thus obtains the correct number of tallies per column.

For the example, the correct subtraction vector is given by (7.14). This gives the last three columns of the tallies in Fig. 7.4. The first two columns of the tallies are not given in the subtraction vector, as they will always be equal to the current value of \( z_j \) being worked on, which for this example is \( z_1 = 3 \). The first two tally columns come
Algorithm 19 Subtraction Vector Adjusting

1: procedure SUBTRACTIONVECTORADJUST($\delta$, $n_z$)
2: \[ s \leftarrow \delta \]
3: for $i \leftarrow 1 : n_z - 1$ do
4: \[ \text{for } j \leftarrow n_z - 1 : -1 : i + 1 \text{ do} \]
5: \[ s_{j-1} \leftarrow s_{j-1} - s_j \]
6: end for
7: end for
8: return $s$
9: end procedure

from the contribution of $n$, and the identity matrix $I$.

\[
s = \begin{bmatrix} 3 & 2 & 1 & 0 \end{bmatrix} \tag{7.14} \]

7.8.1.8 Step 8 - Calculate the Credit Assigned Kernel Value

Using the tally analogy, the credit assigned kernel vector can be calculated using either the row count or the column count. If the column count is used, each column represents a certain subtraction contribution. For example, using the tally example, the first row represents a value of $\frac{1}{5}$. The same information could be represented by performing the subtraction $1 - \frac{1}{2} = \frac{1}{2}$, which gives the first subtraction contribution. Adding a third tally, gives a difference of $\frac{1}{2} - \frac{1}{3} = \frac{1}{6}$ which is the second subtraction contribution.

The subtraction contribution fractions are calculated as follows. Consider a single row in the tally analogy. Starting with one tally, the result is one. For two tallies, the result is $\frac{1}{2}$. Thus the difference between the two results is $1 - \frac{1}{2} = \frac{1}{2}$, which gives the first subtraction contribution. Adding a third tally, gives a difference of $\frac{1}{2} - \frac{1}{3} = \frac{1}{6}$ which is the second subtraction contribution.

Now, by generalizing for any number of tallies and taking into account the value of the credit assignment strength parameter $\lambda$, each column will contribute a subtraction of $\left( \frac{1}{1 + \lambda} - \frac{1}{1 + \lambda(1 + i)} \right)$, where $i$ is the column count starting from zero, at the second column in Fig. 7.4.

Since the first and second columns will always have a count of $z_j$ tallies, they can be handled easily outside of the subtraction vector. The first column is subtracted from the second column, and can be simplified as shown in (7.15).

The remaining column contributions can be then be added together and then subtracted from (7.15) to calculate the credit assigned kernel vector. This equation is
shown in (7.16).

\[ z_j - \left( \frac{z_j}{1 + \lambda \times 0} - \frac{z_j}{1 + \lambda(1 + 0)} \right) = z_j - \frac{z_j}{\lambda + 1} = \frac{z_j}{\lambda + 1} \]  

(7.15)

\[ \hat{k}_j = \frac{z_j}{\lambda + 1} - \sum_{i=1}^{n_x} s_i \left( \frac{1}{1 + \lambda t} - \frac{1}{1 + \lambda(1 + i)} \right) \]  

(7.16)

For the example, for \( \lambda = 1 \), the value of \( \hat{k}_1 \) is calculated using (7.16) in (7.17).

\[ \hat{k}_1 = \frac{3}{2} - 3 \times \frac{1}{6} - 2 \times \frac{1}{12} - 1 \times \frac{1}{20} = 0.78333 \]  

(7.17)

7.8.1.9 Step 9 - Repeat and Finalize

Repeat from step five, until all credit assigned kernel vector entries have been calculated. The calculation of the remainder of the \( \hat{k} \) values is left as an exercise for the interested reader.

Finally, the last entry in the kernel vector will be the value of \( k \). Copy this value to the variable \( k \), then remove it from the kernel vector.

7.8.2 Multidimensional Continuous Credit Assignment

Generalizing the combinatorics based credit assignment algorithm to multidimensional input vectors is simple. Each dimension is first treated separately from step two onwards, and the combination tables are calculated using steps two to four for each dimension individually. Next, the combination tables for each dimension are multiplied together element by element to create a single multidimensional combination table. The algorithm can then continue as normal from step five using the new multidimensional combination table, where the value of interest will be selected from the multidimensional combination table.

7.8.3 Credit Assigned Continuous IKRLS-CMAC Algorithm

In Algorithm 20 the credit assigned continuous IKRLS-CMAC algorithm is shown. Where \( \hat{d} \) is a vector storing the desired values for all dictionary values. The desired value can be approximated from an online CMAC where the error is given by using (7.18). The function GetCAKV is not shown explicitly due to its complexity, but can be implemented using the description given by in Section 7.8.1.

\[ d = e + k^T \alpha \]  

(7.18)
Algorithm 20 Continuous REG-IKRLS-CMAC Algorithm

1: procedure CONT-REG-IKRLS-CMAC(i, d)
2: \[ K^{-1} \leftarrow \frac{\lambda}{m}, \quad \alpha \leftarrow \left[ \frac{d_1}{m} \right], \quad P \leftarrow 1, N \leftarrow n_1, \quad d \leftarrow d_1 \]
3: \hspace{1em} for \( t \leftarrow 2, 3 \ldots n_t \) do
4: \hspace{2em} n \leftarrow \text{Normalize}(i_t)
5: \hspace{2em} \hat{N} \leftarrow \begin{bmatrix} N \end{bmatrix}_{n_t}
6: \hspace{2em} \hat{k} \leftarrow \text{GetCAKV}(\hat{N}, u)
7: \hspace{2em} k \leftarrow \hat{k}_{n_{\alpha^{-1}}}
8: \hspace{2em} \hat{k} \leftarrow \hat{k}_{1:n_{\alpha}}
9: \hspace{2em} a \leftarrow K^{-1}\hat{k}
10: \hspace{2em} e \leftarrow d_t - \hat{k}^T \alpha
11: \hspace{2em} \delta \leftarrow k - \hat{k}^T a
12: \hspace{2em} if \( \delta > \nu \) then
13: \hspace{3em} N \leftarrow \hat{N}
14: \hspace{3em} \hat{d} \leftarrow \begin{bmatrix} \hat{d} \end{bmatrix}_{n_t}
15: \hspace{3em} K \leftarrow \begin{bmatrix} K \end{bmatrix}
16: \hspace{3em} \text{for } i \leftarrow 1 : n_{\alpha} \text{ do}
17: \hspace{4em} \hat{k} \leftarrow \text{GetCAKV}(N, N_i)
18: \hspace{4em} K \leftarrow \begin{bmatrix} K \hat{k} \end{bmatrix}
19: \hspace{3em} \text{end for}
20: \hspace{3em} \alpha \leftarrow K^{-1}\hat{d}
21: \hspace{3em} P \leftarrow \begin{bmatrix} P \ 0 \end{bmatrix}
22: \hspace{3em} \text{else}
23: \hspace{4em} v \leftarrow \frac{Pa}{1 + a^TPa}
24: \hspace{4em} P \leftarrow P - va^TP
25: \hspace{4em} \alpha \leftarrow \alpha - K^{-1}ve_c
26: \hspace{3em} \text{end if}
27: \hspace{2em} \text{end for}
28: \text{end procedure}
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

7.8.4 The Problem with Combinatorics Credit Assignment

One large problem with this credit assignment technique is that a method to update $K^{-1}$ and $\alpha$ with credit assignment information without fully recalculating them from scratch has not yet been discovered by the author. Not only is fully recalculating from scratch computationally demanding, but also any information from the weight update algorithm is lost. This makes this credit assignment method incompatible with eligibility, as eligibility information is stored in the kernel weights. Finding a method to update $K^{-1}$ and $\alpha$ efficiently may be suitable future work.

Additionally, another problem is that as $n_z$ increases, the computational complexity explodes as the complexity for generating the combinations table is given by $O(2^{n_z})$, since the number of rows required in the combination table is $2^{n_z} - 1$. The value of $n_z$ rises the more closely spaced training points are, and the more relaxed the sparsification parameter is set. In the authors previous work [106], an approximate algorithm was investigated which reduced the computational complexity to $O(n_z^2)$, at the expense of credit assignment accuracy.

In the remainder of this thesis, the continuous credit assigned KRLS-CMAC is not used or tested in any problems due to these unresolved issues. At the moment, this method is best suited to offline learning problems.

7.9 Continuous CMAC Vector Eligibility

A method for implementing vector eligibility into the continuous CMAC will be presented in this section.

The eligibility algorithm used in the quantized IKRLS-CMAC relies on the fact that there are a discrete number of feature space weights that can be activated and thus tracked for decay. However, in the continuous CMAC, the number of feature space weights is effectively infinite, thus a new method is required.

7.9.1 Continuous CMAC Vector Eligibility Derivation

The current eligibility association vector can be calculated directly by using (6.1) as shown in (7.19), where $x(n_0)$ is the first association vector input into the CMAC, and $x(n_t)$ is the latest association vector.

$$\bar{x} = cA^1bx(n_0) + \ldots + cA^1bx(n_1) + cA^0bx(n_t)$$  \hspace{1cm} (7.19)
The eligibility association vector can be used to create the kernel eligibility vector as in the quantized CMAC with (7.20).

\[ \tilde{k} = \tilde{X} \tilde{x} \]  

(7.20)

To get the eligibility kernel vector using the original kernel vector, (7.19) can be substituted into (7.20) to get (7.21). Then, note that since \( cA^ib \) is scalar, equation (7.22) can be obtained.

\[ \tilde{k} = \tilde{X}x(n_0)cA^ib + \ldots + \tilde{X}x(n_1)cA^ib + \tilde{X}x(n_l)cA^0b \]  

(7.21)

\[ \tilde{k} = k(n_0)cA^ib + \ldots + k(n_1)cA^ib + k(n_l)cA^0b \]  

(7.22)

Since (7.22) uses the original kernel vectors, they can first be generated using the direct kernel vector generation algorithm, and then eligibility can be applied after without the use of the association vectors.

### 7.9.1.1 Continuous Vector Eligibility Example

In (7.23) the eligibility parameters for this example are shown. For simplicity scalar eligibility, or vector eligibility with \( o_e = 1 \) is used, but this method can be used with any eligibility order.

\[ A = 0.8, \quad b = 0.1, \quad c = 1 \]  

(7.23)

First, a quantized eligibility example is shown for comparison. The association vector inputs for \( t = 0, 1, 2 \) in this example are given by (7.24), (7.25) and (7.26). Assume these association vectors are not going to be added to the dictionary for this example.

\[ x(n_0) = [1 \quad 0 \quad 1 \quad 0]^T \]  

(7.24)

\[ x(n_1) = [0 \quad 1 \quad 1 \quad 0]^T \]  

(7.25)

\[ x(n_2) = [1 \quad 0 \quad 0 \quad 1]^T \]  

(7.26)

The dictionary is also pre-populated with association vectors shown in (7.27).

\[ X = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \]  

(7.27)

119
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

First, for comparison, \( \bar{x} \) for time \( t = 2 \) can be calculated by using (7.19) as shown in (7.28).

\[
\bar{x} = cA^2bx(n_0) + cA^1bx(n_1) + cA^0bx(n_2)
\]

\[
= \begin{bmatrix}
0.081 \\
0 \\
0.081
\end{bmatrix}
+ \begin{bmatrix}
0 \\
0.09 \\
0.09
\end{bmatrix}
+ \begin{bmatrix}
0.1 \\
0 \\
0.1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.271 \\
0.09 \\
0.171 \\
0.1
\end{bmatrix}
\] (7.28)

Then by using \( \bar{x} \) in the kernel equation the result in (7.29) is obtained.

\[
\bar{k} = X\bar{x} = \begin{bmatrix}
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
0.271 \\
0.09 \\
0.171 \\
0.1
\end{bmatrix}
= \begin{bmatrix}
0.261 \\
0.19
\end{bmatrix}
\] (7.29)

Now for the direct method, first, each individual original kernel vector is calculated in (7.30), (7.31) and (7.32).

\[
k(n_0) = \begin{bmatrix}
1 \\
0
\end{bmatrix}
\] (7.30)

\[
k(n_1) = \begin{bmatrix}
2 \\
1
\end{bmatrix}
\] (7.31)

\[
k(n_2) = \begin{bmatrix}
0 \\
1
\end{bmatrix}
\] (7.32)

The eligibility kernel vector can then be calculated using the method in (7.22) as shown in (7.33), which produces the same result as in (7.29).

\[
\bar{k} = k(n_0)cA^2b + k(n_1)cA^1b + k(n_2)cA^0b
\]

\[
= \begin{bmatrix}
0.081 \\
0
\end{bmatrix}
+ \begin{bmatrix}
0.18 \\
0.09
\end{bmatrix}
+ \begin{bmatrix}
0 \\
0.1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.261 \\
0.19
\end{bmatrix}
\] (7.33)

7.9.2 Continuous Vector Eligibility Implementation

Continuous vector eligibility can be implemented iteratively by maintaining an up to date matrix of the latest \( \psi \) original kernel vectors that have been used. This matrix is defined as the recent kernel matrix, \( \bar{K} \). Every iteration the latest original kernel vector
is appended to the top of $\tilde{K}$ as is shown in (7.34).

$$
\tilde{K} \leftarrow \begin{bmatrix} k^T \\ \tilde{K} \end{bmatrix}
$$

(7.34)

The entire recent kernel matrix needs to be updated whenever the dictionary grows, as each kernel vector already stored in the matrix will have grown by one value.

In (7.35) the example recent kernel matrix $\tilde{K}$ is grown by the dictionary value, $k(n_3)$, where $n_3$ is added to the dictionary. Thus, the extra column denoted by the *'s must also be updated by the corresponding $k_3(n_i)$ values.

$$
\tilde{K} = \begin{bmatrix} k_1(n_3) & k_2(n_3) & k_3(n_3) \\ k_1(n_2) & k_2(n_2) & * \\ k_1(n_1) & k_2(n_1) & * \\ k_1(n_0) & k_2(n_0) & * \end{bmatrix}
$$

(7.35)

In order to update the recent kernel matrix, the $\psi$ most recent normalized input vectors must also be kept in a matrix defined as $\mathcal{N}$. Then, the * values can be calculated by applying the direct kernel vector equation between the recent normalized input vectors in $\mathcal{N}$ and the latest dictionary value, $n$.

Additionally, a matrix of vector eligibility multipliers must also be maintained. These multipliers are equivalent to the $cA^Tb$ part of (7.22). To update the eligibility multipliers a matrix $M$ is kept. Every iteration $M$ is updated via (7.36), and then appended with $b$ as shown in (7.37). Additionally, a matrix of vector eligibility multipliers must also be maintained. These multipliers are equivalent to the $cA^Tb$ part of (7.22). To update the eligibility multipliers a matrix $M$ is kept. Every iteration $M$ is updated via (7.36), and then appended with $b$ as shown in (7.37)

$$
M \leftarrow MA^T
$$

(7.36)

$$
M \leftarrow \begin{bmatrix} b^T \\ M \end{bmatrix}
$$

(7.37)

When the number of entries in $\tilde{K}/M/\mathcal{N}$ reach the eligibility decay threshold $\psi$, the oldest entry is deleted, as its eligibility multiplier will have decayed to a near zero value.

The actual eligibility kernel vector can be calculated by first calculating the $cA^Tb$ multipliers with (7.38).

$$
v = Mc^T
$$

(7.38)
And then adding up the kernel vectors multiplied by the multipliers in (7.39)

\[ \bar{k} = \sum_{i=1}^{\text{len}(v)} \tilde{K}_{i,v} \]  

(7.39)

7.9.3 Continuous Eligibility Algorithm

In Algorithm 21 the continuous eligibility algorithm is presented and in Algorithm 22 the continuous eligibility update algorithm is shown.

With this continuous method the eligibility computation time becomes dependent on the decay period \( \psi \) of the impulse response. Long decay times will be computationally slow due to equation (7.39), which will require \( \psi \) multiplications and summations every iteration, and also due to (7.36) which is an \((\psi \times o_c) \times (o_c \times o_e)\) matrix multiplication. Fortunately, in most cases the entire decay profile is not required. A small sample of the beginning of the impulse response is all that is needed, as will be seen in the online motion control experiments performed in the following chapters.

A continuous CMAC with eligibility will be denoted henceforth as a \( C_{0,k}E-KRLS-CMAC \). Where \( o_k \) is the order of the b-spline kernel used, and the E represents the use of vector eligibility. For example, a third order b-spline continuous CMAC with eligibility is denoted as the \( C3E-IKRLS-CMAC \), or just \( C3E-CMAC \) when the context is clear.

7.10 Advantages and Disadvantages of the IKRLS-CMAC

7.10.1 Gains and Losses for Different Tasks

The main purpose of the IKRLS-CMAC controller is for real time online motion control learning tasks, however it can be used for more conventional tasks such as regression modeling, time series or system identification. The main advantage when using the IKRLS-CMAC in regression modeling tasks is that learning can be performed in only one training run, compared to the hundreds or thousands of iterations required with the standard CMAC. The effect of credit assignment will also improve modeling error significantly. However, the main loss associated with the IKRLS-CMAC for regression modeling is the increase in computational complexity required for very large dimensionality data. For very large dimensional data the LMS-CMAC will be the only feasible solution in terms of computational power. But for very large dimensionalities the CMAC performs poorly anyway unless the full overlay is used, which then the LMS-CMAC
requires prohibitive memory requirements. For time series and system identification the same gains and losses as with regression apply.

For motion control learning the IKRLS-CMAC will outperform the standard CMAC in almost all cases. The only time the IKRLS-CMAC may not perform as well is with a fast time varying system where the model continually changes. To make the IKRLS-CMAC adaptive for this type of system a sliding window method would need to be utilized. The other disadvantage to the IKRLS-CMAC over the standard CMAC is again the increase in computational complexity, but most motion control systems that are used in feedback error learning control will not use more than four inputs.

7.10.2 Cons of IKRLS-CMAC Methods Compared with Other Methods

While the IKRLS-CMAC has many potential applications, there are still a few faults in the method where other methods will outperform the IKRLS-CMAC. The main fault in the IKRLS-CMAC method is that there is not yet any way to prove stability of the system when used in motion control learning. Other works that use CMACs in online control base their work heavily on Lyapunov Stability theorem such as the work in [107] in order to generate a system where it is provable that the system will converge to a stable state. However, most of these methods require that there be some sort of linearized model which may not be available or possible to calculate.

Another disadvantage to the IKRLS-CMAC is that there is no specified bound on the number of weights that can be generated. As the number of weights increases the computational efficiency of the algorithm is reduced. There is a possibility that a system could require more weights than is computationally feasible which would cause the system to fail. One way around this may be to use a fixed size dictionary as in [95] where the CMAC would continually prune the oldest or weights with the least contribution to the system. Pruning by the oldest weights could also lead to a sliding window based IKRLS-CMAC design. One final disadvantage to the IKRLS-CMAC is that no method for adapting the kernel weight centers has been found yet. This is mainly due to the increased computational burden that would occur when using the RLS algorithm. Other LMS based CMACs and similar algorithms such as DENFIS have adaptive centers implemented.
7. THE QUANTIZED AND CONTINUOUS KRLS-CMACS

7.11 Conclusion

This chapter first introduced the quantized IKRLS-CMAC, which is a CMAC with all the improvements discussed and derived in previous chapters combined into a single algorithm. Then a method to achieve a continuous KRLS-CMAC which is equivalent to a quantized CMAC with infinite resolution and the full overlay was introduced. The continuous CMAC allows for a more efficient implementation of a full overlay quantized CMAC, but is more computationally complex than a quantized CMAC with uniform overlay.

CMAC enhancements applied to the quantized IKRLS-CMAC, such as higher order sensitivity functions, credit assignment and vector eligibility were also implemented into the continuous KRLS-CMAC. These enhancements brought significantly more computational complexity to the continuous IKRLS-CMAC algorithm than they did to the quantized IKRLS-CMAC.

The method for credit assignment is especially computationally complex, and is unlikely to be able to be utilized in an online situation. However, it provides proof of concept, and perhaps another perspective on how the credit assigned CMAC works.

Vector eligibility was also successfully integrated into the continuous IKRLS-CMAC, and it was found that the computational complexity became dependent on the decay parameter $\psi$. 

124
Algorithm 21 Continuous Eligibility Algorithm

1: procedure CONT-KRLS-CMAC(i, d)
2: \[ K^{-1} \left\lfloor \frac{\lambda}{m} \right\rfloor, \alpha \leftarrow e_{gi}, P \leftarrow 1, N \leftarrow n_1, M \leftarrow [], \tilde{N} \leftarrow [], \tilde{K} \leftarrow [] \]
3: for \( t \leftarrow 2, 3 \ldots n_t \) do
4: \( n \leftarrow \text{Normalize}(i_t) \)
5: \( k \leftarrow \text{GetKV}(N, n) \)
6: \( k \leftarrow m \)
7: \[ \begin{bmatrix} M & \tilde{N} & \tilde{K} \end{bmatrix} \leftarrow \text{ContEliUpdate}(\text{ref}_t, \text{ref}_{t-1}, A, b, \tilde{K}, \tilde{N}, M, k, n) \]
8: \( a \leftarrow K^{-1}k \)
9: \( \delta \leftarrow k - k^\top a \)
10: if \( \delta > \nu \) then
11: \[ N \leftarrow \begin{bmatrix} N \\ n^\top \end{bmatrix} \]
12: \( K^{-1} \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta K^{-1} + aa^\top & -a^\top \\ -a^\top & 1 \end{bmatrix} \)
13: \( \alpha \leftarrow \frac{1}{\delta} \begin{bmatrix} \delta \alpha - ae_{gi} \\ e_{gi} \end{bmatrix} \)
14: \[ P \leftarrow \begin{bmatrix} P \\ 0 \\ 0 \\ 1 \end{bmatrix} \]
15: \( k_s \leftarrow \text{GetKV}(n, \tilde{N}) \)
16: \( \tilde{K} \leftarrow \begin{bmatrix} \tilde{K} \\ k_s \end{bmatrix} \)
17: else
18: \( \nu \leftarrow Mc_i^\top \)
19: \( \tilde{k} \leftarrow \sum_{i=1}^{\text{len}(\nu)} \tilde{K}_{i, \nu_i} \)
20: \( a \leftarrow K^{-1}\tilde{k} \)
21: \( n \leftarrow \frac{Pa}{1 + a^\top Pa} \)
22: \( P \leftarrow P - na^\top P^\top \)
23: \( \alpha \leftarrow \alpha - K^{-1}n e_{gi} \)
24: end if
25: end for
26: end procedure
Algorithm 22 Eligibility Update

1: procedure CONT_ELIG_UPDATE(ref_t, ref_t−1, A, b, K, Ů, M, k, n)
2:  if ref_t != ref_t−1 then
3:      M ← []
4:      Ů ← []
5:      K ← []
6:  end if
7:  M ← MAᵀ
8:  M ← \[
9:      \begin{bmatrix}
10:      n & \n
11:      \end{bmatrix}
12:   \]
13:  K ← \[
14:      \begin{bmatrix}
15:      k
16:      \end{bmatrix}
17:  \]
18:  n ← len(K)
19:  if n > ψ then
20:     M₁,₂ ← []
21:     Ů₁,₂ ← []
22:     K₁,₂ ← []
23:  end if
24: end procedure
8

Regression Tests

In this chapter the CMACs developed earlier and several already established CMACs will be compared using regression tests. As these tests are not for online motion control, vector eligibility will not be used. The results on association vector normalization, and the results on minimum vs. radial sensitivity functions obtained in Chapter 4 will also be confirmed using a regression test problem.

8.1 Testing Method

In the literature many neural networks are compared against one another simply by tweaking the parameters and searching manually for the best results. However, this introduces bias into the results as it is most likely that the creator of the algorithm being compared will have a greater intuition on the tweaking of their own algorithm parameters. In this work, each CMAC variant is tested over a wide range of values for the generalization ratio, sparsification parameter and learning rate parameters. Each CMAC variant may have a range of parameter values that are optimal. Simply averaging all the collected error data together would give inaccurate results as some CMACs could be tested over a greater optimal range than others. To remove this bias, the error data is grouped into a common parameter, which for the CMAC is the generalization ratio. The minimum over all results for each generalization ratio tested is taken, and then these minimums are analyzed and used to draw comparisons between the CMACs.

In each test, and for all the testing chapters, a mid-range Intel i5 750 4-core CPU clocked at 2.67 GHz is used to run the experiments.
8. Regression Tests

8.2 Resolution Effects

In multidimensional CMACs the quantizing resolution controls three aspects of performance. The first is that low quantizing resolution will cause multiple input space states to share the same weights. Using a higher order basis function can help as it will smooth out these shared areas through interpolation, however fine variations may still be lost. The second is that low quantizing resolution with the uniform overlay will cause a reduced number of feature space weights, causing the CMAC to be less of a universal approximator. This means that perfect recall of the training data will not be possible and the least squares approximation will be poorer. Finally, the approximation of the uniform overlay to the full overlay will affect the interpolation accuracy of the CMAC. In the one dimensional CMAC the quantizing resolution controls only one performance aspect of the CMAC which is the effect where low quantizing resolution causes too many input space states to share the same weights. These three issues are tested in this chapter.

In these tests the LMS-CMAC will always use the linear sensitivity function, and the SOCM will always use the Gaussian kernel function.

8.3 Minimum vs. Radial and Normalization Sensitivity Function Tests

In these tests the multidimensional sensitivity function is tested with the radial multiplication based method, and the minimum based method which is suggested in [9]. In [9] it is suggested that the minimum based combination function is superior, however it was never fully investigated or tested against a range of generalization parameter values. Here it is hypothesized that the value of $h$ plays a large role in the optimal choice of the combination function for regression due to the results obtained in Chapter 4.

Each ablative test consists of averaging a large sample of CMAC errors for a random training set calculated from the sombrero function for various values of $h$. A multidimensional sombrero function is defined by (8.1), where $n_d$ is the number of dimensions of the sombrero function desired.

$$f(i_1, i_2, \ldots i_{n_d}) = \begin{cases} 
8 \sin \left( \frac{\sqrt{i_1^2 + i_2^2 + \ldots + i_{n_d}^2}}{\sqrt{i_1^2 + i_2^2 + \ldots + i_{n_d}^2} + i_{n_d}} \right) & \text{for } i_1^2 + i_2^2 + \ldots + i_{n_d} \neq 0 \\
8 & \text{otherwise}
\end{cases} \quad (8.1)$$
8.3 Minimum vs. Radial and Normalization Sensitivity Function Tests

In these tests, \( h \) is tested over the range \( h = 5 \) to \( h = 40 \). For each test the values of \( \phi \) and \( \nu \) are iterated for each value of \( h \) over the ranges \([0.1 : 0.02 : 0.4]\) and \([0.1 : 0.1 : 0.9]\) respectively. The CMACs were trained on a set of 1000 uniformly generated random points, where the same training set was used for each test. Linear sensitivity was used for each test and each CMAC did not use credit assignment, however credit assignment should not affect the comparison of these results. The two dimensional CMAC output was sampled with a distance of 1, the three dimensional CMAC with a distance of 2 and the four dimensional CMAC with a distance of 2.5. The error results for the uniform overlay are divided by the error given by the continuous full overlay C3-CMAC which gives a comparison ratio, where a value of unity is a perfect match, and anything larger than one is not. The results are then plotted and analyzed.

8.3.1 Two Dimensional Results

The results of the two dimensional comparison tests over various generalization parameter values are shown in Fig. 8.1. These results show that the normalized linear radial function consistently produced the lowest test error as expected from the kernel error results obtained in Chapter 4. The non-normalized linear radial function consistently produced results that were slightly poorer in comparison.

Both minimum based sensitivity functions performed poorly in comparison to the radial functions, which is expected as the work in [9] states that improvements from the minimum sensitivity function were only observed in CMACs of input dimensionality greater than two.

A value of around \( h = 10 \) is the beginning of stable convergence between the various tested sensitivity functions.

8.3.2 Three Dimensional Results

The three dimensional results are shown in Fig. 8.2. The large spikes in the full overlay approximation error stop at around \( h = 13 \). For the non-normalized results where \( h < 13 \), the results show that the minimum function was the better approximator. For values \( h \geq 13 \), the radial function became the better approximator.

However, the normalized radial function outperformed all the other combination functions.
Figure 8.1: 2D Combination Function Comparison Tests - Key: Radial (—); Radial (No Norm) (---); Minimum (----); Minimum (No Norm) (-----).

Figure 8.2: 3D Combination Function Comparison Tests - Key: Radial (—); Radial (No Norm) (---); Minimum (----); Minimum (No Norm) (-----).
Figure 8.3: 4D Combination Function Comparison Tests - Key: Radial (--); Radial (No Norm) (--); Minimum (----); Minimum (No Norm) (-----).

8.3.3 Four Dimensional Results

The four dimensional results are shown in Fig. 8.3 and they show that the normalized radial sensitivity function was the best performer over almost all generalization parameters.

The non-normalized radial function was the poorest performer especially when the generalization parameter was below around $h = 20$. The non-normalized minimum sensitivity performed much better when compared to the non-normalized radial. This confirms the findings in [9] where it is stated that the minimum function performed better.

However, normalization of the radial function produces results that are much better than either the normalized or non-normalized minimum functions.

Also, note that as the dimensionality increases, even for the largest generalization parameters tested, the approximation ratio begins to converge on larger values. For example for the four dimensional IKRLS-CMAC, the best ratio was around 1.05. This shows that as the dimensionality increases, near perfect kernel approximation is unattainable for reasonable generalization parameter sizes.
8. REGRESSION TESTS

8.4 Comparisons and Correlations with the Kernel Error

Figures 8.4, 8.5 and 8.7 show the normalized linear radial kernel error and normalized linear regression error graphs together on two separate axes. In these graphs the two dimensional CMAC was trained with 1000 training points and the three dimensional CMAC with 2000 training points. They both used an input range of \( i_{\text{max}} = 10 \) and \( i_{\text{min}} = 10 \) in all dimensions. For the four dimensional CMAC 3000 training points were used and the range was changed to \( i_{\text{max}} = 5 \) and \( i_{\text{min}} = -5 \) in all dimensions so that less data was required for training. However, still for \( \phi = 0.1 \), 3000 training points were not enough to sufficiently train the system and thus those results are not shown in the graph.

These graphs show that the fluctuations in kernel approximation error and regression error are correlated very closely for all dimensionalities. This indicates that the full overlay kernel approximation error plays a large role in determining the CMAC output error in a real learning situation.

Also in these figures the universal approximation ratio plot is shown. This ratio is the number of training inputs divided by the number of feature space weights. For larger generalization ratios the resolution is smaller and thus the number of feature space weights is much less. This means that the CMAC is less able to exactly recall a learned training set. By increasing the generalization parameter \( h \), the number of feature space weights is increased, but the number of training points remains constant, and thus the universal approximation ratio is reduced. The graph shows how larger generalization ratios follow the universal approximation ratio plot more closely. For small values of the generalization ratio \( \phi \), the approximation to the full overlay error is more dominant as the feature space weights are sufficient.

In conclusion, larger generalization parameters improve modeling performance due to the well known universal approximation problem in multidimensional CMACs being improved, and also because the full overlay approximation becomes better. However, this increase comes at the cost of increased computation time which will be explored in the online motion control experiments.

8.5 1D Sine Wave Regression Comparison Test

In this test the IKRLS-CMACs regression performance is tested on a one dimensional sine wave. The one dimensional CMAC is considered to be different to the multi-
Figure 8.4: 2D Regression Correlation - Correlation between the regression error, the kernel approximation error and the universal approximation error. **Key:** Regression Error $\phi = 0.25$ (---); Regression Error $\phi = 0.1$ (-- -- --); Kernel Approximation Error (---); Universal Approximation Error $\phi = 0.25$ (---); Universal Approximation Error $\phi = 0.1$ (---).

Figure 8.5: 3D Regression Correlation - Correlation between the regression error, the kernel approximation error and the universal approximation error. **Key:** Regression Error $\phi = 0.25$ (---); Regression Error $\phi = 0.1$ (-- -- --); Kernel Approximation Error (---); Universal Approximation Error $\phi = 0.25$ (---); Universal Approximation Error $\phi = 0.1$ (---).
8. REGRESSION TESTS

Figure 8.6: 4D Regression Correlation - Correlation between the regression error, the kernel approximation error and the universal approximation error. Key: Regression Error $\phi = 0.25$ (—); Kernel Approximation Error (· · ·); Universal Approximation Error $\phi = 0.25$ (—).

Figure 8.7: 4D Regression Correlation - Correlation between the regression error, the kernel approximation error and the universal approximation error. Key: Regression Error $\phi = 0.25$ (—); Kernel Approximation Error (· · ·); Universal Approximation Error $\phi = 0.25$ (—).
8.5 1D Sine Wave Regression Comparison Test

multidimensional CMAC since it is a universal approximator, as explained in Section 1.5.3.2.

In Fig. 8.8 the results for the IKRLS-CMACs are shown for a test that was tested over a range of $[0.1 : 0.01 : 0.25]$ for $\phi$ and $[0.1 : 0.05 : 0.9]$ for $\nu$. The LMS-CMAC is iterated over the same generalization ratio range, and the LMS learning rate is iterated over $[0.1 : 0.01 : 0.9]$. The SOCM is also iterated over the same generalization ratios, but the sparsification parameter is instead iterated over $[0.1 : 0.1 : 0.9]$ and the learning rate is instead iterated over $[0.1 : 0.1 : 0.9]$ for computational reasons, although it was ensured that the difference in errors between learning rates was not large enough to cause concern. The quantization/normalization bounds are given by $i_{\text{max}} = 10$ and $i_{\text{min}} = -10$, and a set of 1000 randomly generated training points was used in all tests. All CMACs used a generalization parameter of $h = 10$ unless otherwise specified.

The tests involving the linear and order two sensitivity functions had their generalization ratio $\phi$ value normalized for local generalization area size by equations (4.5) and (4.7) respectively in order to normalize the local generalization areas for a fair test, as explained in Section 4.3.1.

8.5.1 Results

Although the approximation to the full overlay is not a problem in the 1D CMAC, the B-CMAC and CAB-CMAC performed very poorly. This is because the quantization resolution is not large enough when $h = 10$ is used.

The linear sensitivity based L-CMAC was able to greatly reduce the modeling error. Credit assignment further reduced the error and the CA2L-CMAC performed only slightly poorer when compared to the CAL-CMAC. The CALh25-CMAC is a CAL-CMAC where $h = 25$. It performs better than the CAL-CMAC due to its increased generalization parameter. The O2-CMAC performs slightly better than the L-CMAC, and the CAO2-CMAC performs better than the CAL-CMAC.

The C0-CMAC performed much better than the B-CMAC and CAB-CMAC due to its infinite resolution. The C3-CMAC however only performed slightly better overall compared to the L-CMAC, which shows how well the linear based sensitivity reduces low resolution issues for quantized CMACs.

The LMS-CMAC used the linear sensitivity overlay and competed well with the higher order sensitivity based IKRLS-CMACs. The SOCM on the other hand with its Gaussian kernel performed much more poorly.
8. REGRESSION TESTS

Figure 8.8: 1D CMAC Ablative Tests - Comparisons between various CMAC implementations.

8.6 Multidimensional Sombrero Function Regression Tests

In this section the CMAC variants developed in this work, and the LMS and SOCM CMACs are tested on a two to four dimensional sombrero regression modeling problem. The same method used in the one dimensional test and described in the introduction of this chapter is used to gather and analyze the data.

The same range as in the 1D test is used for $i_{\text{max}}$ and $i_{\text{min}}$ and a set of 1000 uniform random training points were used for training. For the 4D CMAC 2000 uniform random training points were used in an attempt to cover the larger input space.

In Figures 8.9, 8.10 and 8.11, the regression error results are shown as box plots. The results show that for all dimensionalities, the binary based sensitivity function performed poorly due to low quantizing resolution, and additionally now in the multidimensional cases due to poor universal approximation power and a poorer approximation to the full overlay. The linear sensitivity based CMACs performed significantly better as they allowed for better least square approximations to the training set and much better approximations to the full overlay.

The second order sensitivity CMACs performed very poorly in comparison especially as the dimensionality was increased. This matches with the statements in [9], where it was stated that the second order sensitivity function performed poorly in the multidimensional CMACs. This is probably because 1) the second order sensitivity
function does not give any improvement in approximation to the full overlay over the linear sensitivity function, and 2) a larger generalization ratio is required to keep the same local generalization area as the linear sensitivity CMAC, meaning lower resolutions and thus poorer universal approximation power.

Credit assignment also reduced the error, but the improvement was not as large as the difference between the binary and linear CMACs. For all dimensionalities the secondary credit assignment method produced error results almost identical to the primary credit assignment method.

Increasing the generalization parameter to $h = 25$ in the two and three dimensional CMACs, and to $h = 29$ in the four dimensional CMAC reduced the error significantly due to both the increased universal approximation capabilities and better approximation to the full overlay.

The continuous CMACs for the two and three dimensional IKRLS-CMACs significantly outperformed the non-credit assigned binary and linear sensitivity quantized IKRLS-CMACs. As the dimensionality increased in the three and four dimensional CMACs, the minimum box plot value of the C3-CMAC gave the lowest error, but the maximum values tended to remain the same as the L-CMAC.

The LMS-CMAC was unable to outperform the IKRLS-CMACs with the same amount of training in any dimensionality. Also, the SOCM was the worst performer, performing only slightly better than the binary IRKLS-CMACs in the two dimensional case and performing much worse than any other CMAC in the three dimensional CMACs. Note that the 4D SOCM was not tested as it was too computationally slow to allow testing in a reasonable period of time.

8.7 Conclusions

In conclusion these regression results show that the value of $h$ is critical in determining the performance of the CMAC. It was also shown that the minimum multidimensional combination function was only better in some cases when the association vector was not normalized and the value of $h$ was low. When the association vector is normalized with the two-norm, the radial function is always the better choice. The results showed that the average regression error is closely correlated with both the average kernel approximation error and the universal approximation ratio.

There is really no need to ever use the minimum function. It is likely that the authors of [9] only used the minimum sensitivity function with small values of the
8. Regression Tests

Figure 8.9: 2D Sombrero Regression Tests - Comparisons between various CMAC implementations.

Figure 8.10: 3D Sombrero Regression Tests - Comparisons between various CMAC implementations.
8.7 Conclusions

![Figure 8.11: 4D Sombrero Regression Tests](image)

**Figure 8.11: 4D Sombrero Regression Tests** - Comparisons between various CMAC implementations.

generalization parameter $h$, and high dimensionalities, and that is why they saw improvements with the minimum sensitivity function. Also, in [9] the authors did not ever normalize the association vector.

Using the results it can be concluded that the linear sensitivity function and credit assignment enhancements lower modeling errors significantly for the KRLS-CMAC. The second order sensitivity function does not perform as well as the linear sensitivity function especially in high dimensional cases. The continuous CMACs can produce the lowest error, however with no credit assignment they cannot perform as well on average as the quantized CMACs with a large generalization parameter.

From these results it is recommended that the association vector use a linear sensitivity function, be normalized with the two-norm, always use the radial function, and the value of $h$ should be chosen as large as is computationally feasible, whilst avoiding the spikes in the uniform overlay kernel approximation graphs shown in Chapter 4. From this point onwards, the minimum combination sensitivity function will be disregarded as an option, and the two-norm normalized radial sensitivity function will always be used.
8. REGRESSION TESTS
9

System Identification Tests

In this test a non-linear time delay system is identified using the multiple CMAC variants developed in this work, and several already well established CMAC algorithms. Equation (9.1) is used to describe a non-linear plant with multiple time delay, where \( f(\sigma) \) is given by (9.2). This is the same plant that is also used in [74] for testing the recurrent fuzzy CMAC. The Recurrent Fuzzy CMAC (RFCMAC) is derived and tested in [74] and is similar to the IKRLS-CMAC in that it combines multiple improvements together. It is uses a gradient descent based training algorithm, computes weights in the kernel space with a continuous Gaussian kernel and uses a fuzzy training scheme. It is also self organizing in that the dictionary values and their corresponding kernel function support width values are automatically adjusted from the data via gradient descent. It also utilizes a recurrent unit which aids in the learning of time delay systems. The RFCMAC is very similar to the SOCM, except for the fact that it uses a recurrent unit and fuzzy calculations.

\[
y(t + 1) = f(y(t), y(t - 1), y(t - 2), u(t), u(t - 1)) \tag{9.1}
\]

\[
f(x_1, x_2, x_3, x_4, x_5) = \frac{x_1x_2x_3x_5(x_3 - 1) + x_4}{1 + x_2^2 + x_3^2} \tag{9.2}
\]

The goal of this experiment is for the CMAC to learn the plant model through a training sequence, then test how well the CMAC is able to model the plant on a test set of data.

All the CMACs are trained over 1000 iterations. For the first 500 iterations a random uniform sequence over the range \([-2, 2]\) is used as the training data. Each experiment used the same set of random data. The data for the remaining 500 iterations is created by a sinusoid given by \(1.05 \sin(\pi/45)\). The input to the CMAC is given by \(u = [u(t + 1) \ y(t + 1)]\). This is the same training scheme used in [74]. The same
testing method used in the regression problems is used here, where the CMACs are trained over a variety of $\phi$ and $\nu$ parameters. The LMS-CMAC used a learning rate range of $[0.05 : 0.01 : 1.5]$ and the SOCM and RFCMAC a range of $[0.1 : 0.05 : 0.95]$. The input sequence used to test the trained CMAC is given by (9.3).

$$u(t + 1) = \begin{cases} \frac{\pi t}{25} & 0 < t < 250 \\ 1.0 & 250 \leq t < 500 \\ -1.0 & 500 \leq t < 750 \\ 0.3 \sin \left( \frac{\pi t}{25} \right) & 750 \leq t < 1000 \\ +0.1 \sin \left( \frac{\pi t}{32} \right) & \\ +0.6 \sin \left( \frac{\pi t}{10} \right) \end{cases}$$  \hspace{1cm} (9.3)

9.1 Results

In Fig. 9.1 the comparison total absolute error test results for the system identification problem are shown as a box plot.

The results show that the B-CMAC was the poorest performer, and the credit assigned CAB-CMAC performed better overall. The L-CMAC performed much better than both binary CMACs as expected, and the credit assigned CAL-CMAC further improved results. The linear sensitivity based CMACs also exhibit greatly reduced spread, especially the CAL-CMAC indicating that credit assignment improved results for poorly performing generalization ratios. The CALh25-CMAC is a CAL-CMAC with generalization parameter set to $h = 25$. It was the best performer in these tests.

The secondary credit assignment method CA2L-CMAC exhibited very similar results to the CAL-CMAC, only performing very slightly more poorly.

The continuous CMACs performed better than the non-credit assigned quantized CMACs as expected, with the C0-CMAC actually performing better in terms of the median and minimum in comparison to the C3-CMAC. However, the continuous CMACs did not outperform the CALh25-CMAC.

The LMS-CMAC performed better than the binary quantized IKRLS-CMACs, but could not outperform the linear based IKRLS-CMACs. The SOCM performed poorly in comparison to all CMACs except the B-CMAC.

Finally, the RFCMAC did not perform as well as expected, having a similar median to the L-CMAC, but also having a very large maximum error which was around 137.
9.2 Qualitative Comparison Against Other Methods

![Figure 9.1: System Identification Test Results - Comparison results for the system identification task.](image)

The RFCMAC was found to perform well over a small range of parameters only, and performed very poorly over the rest. A large amount of parameter combinations also failed to maintain numerical stability, producing numerically indefinite results.

9.2 Qualitative Comparison Against Other Methods

Although the standard CMAC is not often used in system identification as other methods outperform it, the IKRLS-CMAC performs significantly better in comparison to other CMAC models as shown above. However, when compared to other system identification methods that work well with chaotic systems such as adaptive learning fuzzy rule controllers the performance can not expected to be as good. This is because although the IKRLS-CMAC is an excellent local learning algorithm its ability to model areas involving fine changes is not as powerful as other adaptive methods.

The Dynamic Evolving Neural-Fuzzy Inference System (DENFIS) [108] is an adaptive fuzzy system that has found excellent performance in time series prediction and system identification. It is an adaptive fuzzy system that evolves online through incremental hybrid unsupervised/supervised learning. In some ways it is similar to the CMAC in that its output is generated from m activated weights or fuzzy rules in the case of DENFIS. The main advantage the DENFIS system has over the CMAC is that it effectively has a means to move its equivalent kernel centers, which the CMAC does...
9. SYSTEM IDENTIFICATION TESTS

not. This makes its ability to model everything more accurately. The reason adaptive kernel centers was not implemented into the IKRLS-CMAC was because no method to make it compatible with vector eligibility was found and additionally it would significantly increase the computational burden when using the RLS algorithm. However there are other LMS-CMAC variants that incorporate adaptive kernel centers such as the Self Organizing CMAC (SOCM) [25].

9.3 Conclusion

In conclusion this system identification test showed that the IKRLS-CMACs developed in this work outperform the well established CMACs tested in this chapter. The linear sensitivity function and credit assignment enhancements applied to the IKRLS-CMACs significantly improved the results. The best performer was the CALh25-CMAC, which used credit assignment, linear sensitivity and a generalization parameter of $h = 25$. The continuous CMACs developed in this work also performed well, but could not outperform the CALh25-CMAC. The other well established third party CMACs also could not outperform any of the linear sensitivity based IKRLS-CMACs.
10

Motion Control Test - Inverted Pendulum

The typical inverted pendulum is a cart that is constrained to move along a linear railing, where in the middle of the cart is a pole attached by a freely rotating and non-actuated pin joint. The control objective of the inverted pendulum is to maintain the pole in an upright position after a disturbance is applied, by manipulating the position and velocity of the cart along the rail.

This problem has been solved by the CMAC in previous works such as in [6] which used the FOX-CMAC, in [109] where a CMAC was combined with a linear regulator controller, and in a reinforcement learning problem in [87] where two CMACs were used to stabilize the inverted pendulum.

A simulated inverted pendulum is used to test the performance of the CMACs described in this work. The simulated cart mass is 5kg, with a pole mass of 1kg and pole length of 2m. The cart was constrained to allow a maximum of $\pm 50N$ of control force in order to simulate actuator saturation and also a maximum slew rate of $\pm 1000N/s$ to prevent unrealistically fast switching. MATLAB simulink with simmechanics was used to model and simulate the system with ODE3 and a fixed timestep of 0.01s.

In Fig. 10.1 the inverted pendulum simulink model is shown and in Fig. 10.2 the control system for the IKRLS-CMAC and inverted pendulum is shown. This is the same control system utilized by the FOX-CMAC which is shown in Fig. 6.1.

In all experiments, the pole begins balanced in the upright position, and a short pulse of magnitude 80N is applied to the pole in order to disturb it. Here $\theta$ is defined to be the pole angle against the horizontal which is bounded between the angles $[\pi - \pi]$, where an angle of zero indicates that the pole is in the upright position. Similarly, $p_c$
Figure 10.1: Simmechanics Inverted Pendulum - Internal crane implementation for the simmechanics inverted pendulum.

Figure 10.2: Simmechanics Inverted Pendulum Control Scheme - Control scheme for the simmechanics inverted pendulum.
is defined as the cart position along the rail, where $p_c = 0$ is when the cart is in the middle of the railing.

First, a simple proportional feedback controller was very roughly designed to try and stabilize the pendulum. Note that no effort went into optimizing this feedback controller, other than ensuring the pendulum eventually converged or reached a stable oscillatory pattern when a small impulse was applied. The feedback control equation is given by (10.1), where for this problem $\text{ref} = 0$.

$$y = -100(\theta - \text{ref}) \tag{10.1}$$

The input to the CMACs tested is given by $u = \begin{bmatrix} \theta & \frac{d\theta}{dt} \end{bmatrix}$. Here the limits for quantization and normalization are given by $i_{\text{max}} = \begin{bmatrix} 0.1 & 0.5 \end{bmatrix}$ and $i_{\text{min}} = \begin{bmatrix} -0.1 & -0.5 \end{bmatrix}$. The training error for the CMAC is given by $e = \theta - \text{ref}$.

The impulse response was measured by applying a small impulse to the pole pin joint and measuring $\theta$. The response is shown in Fig. 10.3 as the solid line. A second order eligibility profile approximation (see Section 6.2.1) was selected through trial and error, and is given by the values $p_a = 2.5$, $p_b = 1.2$, $p_c = 1$ and $p_t = 0.01$. This eligibility profile is shown in Fig. 10.3 as the dashed line and has a settling time of approximately 1000 timesteps. Due to the stochastic nature of online learning, it can be difficult to determine whether a change to the CMAC algorithm improves the control of a system overall. This is because an error in training early on due to poor modeling can potentially send the system into a state which results in better training overall for a particular set of parameters, even though it may make it worse for most other parameters. To test to see if the improvements actually improve training overall they must be tested over a wide range of parameters. The same testing scheme used in the regression problems described in Section 8.1 is used here for this purpose.

### 10.1 Results

In this section the improvement that is brought to the quantized and continuous IKRLS-CMACs by each enhancement is tested. Furthermore, the CMACs developed in this thesis are compared against other CMAC variants such as the FOX-CMAC, the self organizing CMAC (SOCM) and the LMS-CMAC. All third party CMACs used the linear sensitivity function except for the SOCM which used a Gaussian kernel as specified in [25].
10. MOTION CONTROL TEST - INVERTED PENDULUM

![Graph showing pole angle response over time]

**Figure 10.3: Inverted Pendulum Impulse Response - Key:** The first 1000 timesteps at a simulated sample rate of 0.01s per timestep for the response of the pole angle to an impulse applied to the tip (---); The approximated second order eligibility profile (----).

The tests where carried out over a wide range of parameter values for generalization ratio $\phi$, sparsification parameter $\nu$ and learning rate $g_e$. The value of $\phi$ was iterated over the range $[0.2 : 0.02 : 0.4]$, $\nu$ was iterated over the range $[0.3 : 0.3 : 0.9]$ and $g_e$ was iterated over the range $[1 : 2 : 40]$ for all IKRLS-CMAC experiments. The FOX, SOCM and LMS-CMACs were tested over the learning rate range $[0.0001 : 0.0003 : 0.01]$. The learning rate ranges differ as different CMAC variants have different optimal learning rate ranges. The ranges were determined by hand tuning the CMACs and determining at what learning rate values the CMAC output became too large such that it destabilized the system, or too small to produce a noticeable impact. The decimations were chosen such that the difference in errors between two neighbouring parameter choices was small.

All CMACs with eligibility used the second order eligibility profile shown previously, and an eligibility decay of $\psi = 1000$ unless otherwise stated. The KRLS eligibility weight decay constant $\tau$ was set to $\tau = \frac{1}{50}$ for all experiments. The weight decay constant is not iterated over as it is not particularly sensitive, and $\tau = \frac{1}{50}$ works well in almost all cases. The generalization parameter was set to $h = 10$ for all experiments unless otherwise specified, and all quantized CMACs used the uniform overlay. All
IKRLS based CMACs also used the approximation to $\mathbf{P}$ described in Section 3.4.1.

Note that the FOX and LMS CMACs do not use a sparsification parameter.

For this experiment, the inverted pendulum total absolute error of the pole angle is integrated over the simulation time and this is used as the performance measure. The test was performed over 20s of simulated time.

10.1.1 Comparison Error Results

The comparison results are shown in Fig. 10.4. The results show that the quantized BE-CMAC performed poorly and had the largest median and maximum values of the IKRLS-CMACs. This is because the binary sensitivity function does not give a good kernel response approximation, or provide enough resolution at $h = 10$, even for a two-dimensional CMAC. The LE-CMAC performed significantly better with much a lower median and maximum value.

It is also shown in the results that credit assignment provided a large decrease in the errors in the CALE-CMAC and CA2LE-CMAC experiments. For the CABE-CMAC, credit assignment also decreased the errors. The secondary credit assignment method used in the CA2LE-CMAC was only very slightly inferior compared to the regular credit assignment method.

The C0E-CMAC performed better than the BE-CMAC as expected. However, the C3E-CMAC performed much better than the binary IKRLS-CMACs and was able to perform almost as well as the CALE and CA2LE-CMACs. For the C3E, the range and interquartile ranges were almost identical to the LE-CMAC, however the median was significantly lower for the C3E-CMAC. This may indicate that for the two-dimensional CMAC, the approximation to the full overlay for the linear sensitivity function is fairly good over most values of $\phi$.

The CALE-CMAC with $h = 25$ had errors very similar to when $h = 10$. This is because after $h = 10$, the approximation to the full overlay for the uniform overlay for two dimensions does not improve significantly as can be seen in the results obtained in Chapter 4.

The results show that the FOX-CMAC with linear sensitivity did not have errors as low as any of the IKRLS based CMACs that used linear sensitivity, probably due to its slower learning LMS training algorithm.

Figure 10.4 also shows results for the CAL-CMAC. The learning rate parameters for the CMACs without eligibility are iterated over different ranges compared to the
previous tests. This is because the learning rate ranges that work well with eligibility are different to those that work well without eligibility. The ranges for $\phi$ and $\nu$ were kept the same and $g_c$ was iterated over $[5 : 5 : 100]$.

The CAL-CMAC errors are generally much larger than the IKRLS-CMACs with eligibility, and similar to the FOX-CMAC. However, the results are deceptive and are even worse than what is shown. The second axis of Fig. 10.4 shows the failure rate calculated over all the tests (not just the minimums), where the results are denoted by the gray diamond. If the pendulum angle did not settle to within $\pm 0.05$ radians during the 20s of simulated time, the experiment is marked as a failure. Almost all the CMACs with eligibility were able to successfully stabilize the pendulum within the allocated time for all parameter combinations with a failure rate of less than 10%. However, the CAL-CMAC was not able to stabilize the pendulum successfully in the majority of cases, with its 99% failure rate. The CAL-CMAC tended to simply reduce the oscillation amplitudes without actually settling the pendulum. These results show that the CAL-CMAC is actually unusable for motion control.

These results clearly show that eligibility is actually required to be able to stabilize and control the inverted pendulum, and controlling a system without it is simply a matter of luck in finding a rare parameter set that works. This agrees with [6] which states that the CMAC without eligibility in a feedback error learning configuration is not suited to controlling online motion control systems.

10.1.2 Average Computation Time Results

In Fig. 10.5 the computation time results per iteration for the different ablative and comparison tests are shown. The computation times used are the computation times for the minimum TAE results. Also, in Fig. 10.6 the number of weights used by each CMAC is shown.

The computation time results show that there is little speed difference between the binary and linear sensitivity functions for the quantized CMACs. However, the linear sensitivity CMACs used less weights compared to the binary CMACs. This is probably because stabilization of the pole was smoother with linear sensitivity, and thus the system was pushed into less states which would require weights to be added to the dictionary. This difference in the weights used can explain why the computation times are even, as the linear sensitivity function is expected to be slower due to its increased complexity.
There is a small increase in computation time when credit assignment is used. The main increase in computation time comes from the $K^{-1}$ matrix update procedure. By keeping the generalization parameter $h$ and the number of weights $n$, small through larger sparsification parameter values, this update can remain fast. The secondary credit assignment update method was much faster on average compared with method one credit assignment due to the fact that it does not require an inverse kernel matrix update every iteration. When the generalization parameter is increased to $h = 25$, the computation time increases further, but by an acceptable amount.

The continuous CMACs had the largest computation time. Their computation times are dependent on the eligibility decay time $\psi$ which was set relatively large at $\psi = 1000$ for this experiment. It should be noted that the eligibility decay time is kept constant across all tests to make the error comparison with the other CMACs fair, by ensuring all CMACs have the exact same training parameters. In the next section it will be seen that the computation time can be significantly reduced without sacrificing modeling performance by reducing the decay time.

The FOX-CMAC was tested and the results show that it has the lowest computation time as is expected since it is a LMS based solution. However, the quantized KRLS based CMACs were still competitive. The number of weights used is not shown for the FOX-CMAC as it is not a kernel based solution, and thus the weight vectors are not
Figure 10.5: Inverted Pendulum Computation Times - Computation times for the inverted pendulum tests for the minimum TAE results.

comparable.

The CAL-CMAC operated at a similar but slightly faster computational speed to the CALE-CMAC, indicating that eligibility does not increase the computation time for the quantized CMAC significantly. The CAL-CMAC also used a similar number of weights when compared to the other CMACs.

In conclusion, these results show that the tested two dimensional quantized IKRLS-CMACs are capable of computational speeds that are competitive to the LMS based FOX-CMAC in this inverted pendulum problem.

10.1.3 Eligibility Decay Value Computation Time Tests

In these tests in this section the effect of the vector eligibility decay time on the computation time is tested on the quantized and continuous CMACs. The parameters used for the CMACs variants tested were those that gave the best TAE results from the previous tests.

10.1.3.1 Quantized CMAC

In Fig. 10.7 the computation times and average best settling times for various values of eligibility decay times $\psi$ are shown. These are the error and computation time results averaged over all tested parameters. The results show that for a decay time
greater than $\psi = 500$ there is no noticeable difference in pole angle error. Additionally, the graph also shows the average computation times per iteration for each decay time tested. This shows that the quantized IKRLS-CMACs with eligibility have a constant computation time regardless of the eligibility decay time.

### 10.1.3.2 Continuous CMAC

In Fig. 10.8 the same test as in Section 10.1.3.1 is performed, but with the continuous CMACs. The results show that as the decay time increases, the computation time per iteration also increases in a near linear way. The settling time error is also stable from about $\psi = 100$ decay time steps.

These results show that good computational performance can be obtained with the continuous IKRLS-CMAC with eligibility. Thus, to be fair to the continuous CMAC with eligibility, the computation time results in Section 10.1.2 should read as the time taken for a decay time of around $\psi = 100$, bringing the computation speed down to around 0.6s, which is similar to the computation time of the CALEh25-CMAC.

Furthermore, these results also show that the computation time increase due to use of the third order kernel is constant over the decay time and not overly significant.
Figure 10.7: Computation Times for Quantized Eligibility Ablative Tests - Key:
Average Error (—•); Average computation time in seconds (—*).

Figure 10.8: Eligibility Decay Continuous CMAC Computation Time Results
- Key: C3E-CMAC average settling timesteps (—•); C0E-CMAC average settling
timesteps (—•); C3E-CMAC average computation time (—•); C0E-CMAC average
computation time (—*).
10.1.4 Actual Computation Times

While the average computation times show the differences in performance overall between each CMAC well, they do not show how well the CMACs will perform in a real situation. Here the actual computation times versus the number of weights in the dictionary for the CALE-CMACs is shown for the CMACs that performed well in order to show their applicability to real time online control. Note that the CMAC code was run on MATLAB. It is possible that more optimized CMAC code may be able to be implemented in a more efficient programming language.

In Fig. 10.9 the actual computation times for when a weight is not added to the dictionary is shown, and in Fig. 10.10 the actual computation times for when a weight is added to the dictionary is shown. The computation times are compared against the number of weights in the dictionary. The results show that the situation when a weight is added into the dictionary requires more computation time.

The C3E1000 CMAC variant indicates the case where the decay parameter used is $\psi = 1000$. This parameter choice causes the C3E variant to be very computationally inefficient, as can be seen in the figure, due to the fact that the continuous CMAC algorithm's computational speed is dependent on the size of the decay parameter. With a decay parameter choice of $\psi = 100$ the C3E100-CMAC becomes much faster, but still ends up being slightly slower than theCALEh25-CMAC after a few weights are added into the dictionary.

The CALE-CMAC was slower than the LE-CMAC as expected and the CA2LE-CMAC was faster than the CALE when a weight was not added into the dictionary, but was slower than the CALE when a weight was added. This means the CA2LE-CMAC has a shorter available sampling rate period for online learning, but has fast recall. It may only be useful in situations where the CMAC can be trained offline first.

The maximum number of weights required by the best solutions for the linear sensitivity based CMACs used in the previous tests was 32. In Table 10.1 the computation times for when a weight is added are recorded at $n_{\alpha} = 32$ weights and the maximum sampling rate at this computational speed is calculated. Note that the sample rate chosen will determine the eligibility decay time, as the decay time is based on sampled timesteps. Thus continuous CMACs with high sample rates may have large decay times, causing slow computation. In this test the CMACs were simulated, and sampled at 0.01s or 100 Hz. Sampling at 500 Hz would mean that the minimum decay time would need to be five times greater, so a decay time of $\psi = 500$ would be required.
Table 10.1: Actual Computation Times When $n_o = 32$

<table>
<thead>
<tr>
<th>CMAC Type</th>
<th>Computation Time (ms)</th>
<th>Max Sample Rate (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE</td>
<td>0.5</td>
<td>2000</td>
</tr>
<tr>
<td>CALE</td>
<td>0.6</td>
<td>1667</td>
</tr>
<tr>
<td>CA2LE</td>
<td>0.88</td>
<td>1136</td>
</tr>
<tr>
<td>CALEh29</td>
<td>0.76</td>
<td>1316</td>
</tr>
<tr>
<td>C3E100</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>C3E1000</td>
<td>1.8</td>
<td>555</td>
</tr>
</tbody>
</table>

10.1.5 Eligibility versus no Eligibility CMAC Output Comparisons

A comparison of the inverted pendulum pole angle for a CMAC with and without eligibility is shown in Fig. 10.11. The parameters used were the ones from the error tests which gave the lowest error for the CALE-CMAC and the CAL-CMAC. These parameters were $\phi = 0.446048$ (0.32 binary equivalent), $\nu = 0.3$ and $g_e = 19$ for the CALE-CMAC and $\phi = 0.446048$ (0.32 binary equivalent), $\nu = 0.9$ and $g_e = 76$ for the CAL-CMAC. The graph shows how the CAL-CMAC oscillates whereas the CALE-CMAC controls the pole angle more smoothly. The two horizontal lines around the pole angle of zero indicate the settling tolerance used.

In Fig. 10.12 the differences in requested control effort applied to the cart actuator by the CMAC with eligibility and without eligibility are shown. The CMAC without eligibility needs greater control effort and actually saturates the maximum actuator input quite significantly for the first few cycles. The CMAC with eligibility used a more intelligent control signal which is smooth and does not drive the actuator as hard.

10.2 Comparison with other CMACs

Two other CMAC implementations were tested on the same inverted pendulum problem which were the LMS-CMAC and SOCM. However, these CMACs had median errors far too large to plot on the same graph as the other results. They were also unable to stabilize the pendulum with any parameter set, probably due to a lack of eligibility.

10.3 Effect of the Eligibility Profile

In order to show the effect of using different eligibility profiles that are not a perfect match to the impulse response data, Figures 10.13 and 10.14 are used. Fig. 10.13
Figure 10.9: Actual Computation Times - Actual Computation times for when no weight is added into the dictionary. Key: CALEh25-CMAC (---); LE-CMAC (---); CALE-CMAC (-----); C3E100-CMAC (-----); C3E1000-CMAC (...) CA2LE-CMAC ( ... ).

shows two eligibility profiles that only vaguely attempt to match the impulse response. The solid lines profile is given by $p_a = 9.5$, $p_b = 2.2$, $p_c = 0.35$ and $p_t = 0.01$, and the dashed lines profile is given by $p_a = 1.5$, $p_b = 1.2$, $p_c = 0.15$ and $p_t = 0.01$. In Fig. 10.14 the settling time results for the CMAC with these poorly matched eligibility profiles is shown. The results show that even though the match is poor, the KRLS-CMAC with eligibility is still able to control the system well. This indicates there is a large tolerance in the choice of eligibility profile, which is also indicated by (6.5).

10.4 Conclusion

In this section the various CMAC variants developed in this thesis and from third parties were tested on a simulated inverted pendulum online stabilization problem. The CMACs from this thesis that were tested were the quantized and continuous IKRLS-CMACs. The third party CMACs tested were the LMS, SOCM, and FOX-CMACs.

The results showed that only CMACs with vector eligibility were able to stabilize the pendulum effectively. This meant that all the third party CMACs, with the exception of the FOX-CMAC, were unable to stabilize the pendulum. The CAL-CMAC also could
Figure 10.10: Actual Computation Times - Actual Computation times for when a weight is added into the dictionary. Key: CALEh29-CMAC (—); LE-CMAC (—); CALE-CMAC (——-); C3E100-CMAC (———); C3E1000-CMAC (-----); CA2LE-CMAC (-----).

Figure 10.11: Eligibility Effect on Pole Angle - The difference between a CMAC with eligibility and a CMAC without eligibility when stabilizing the inverted pendulum pole angle. Key: An IKRLS-CMAC with eligibility. (—-); An IKRLS-CMAC without eligibility (———).
Figure 10.12: Eligibility Effect on Control Effort - The difference between a CMAC with eligibility and without eligibility in control effort when stabilizing the inverted pendulum pole angle. Key: An IKRLS-CMAC with eligibility. (—); An IKRLS-CMAC without eligibility (——).

Figure 10.13: Eligibility Profiles Tested - Two poorly matched eligibility profiles which are tested on the inverted pendulum problem. Key: Poorly matched eligibility response one (—); Poorly matched eligibility response two (——); Good eligibility response (⋯); Impulse Response (—).
not stabilized the pendulum as witnessed by the fact that the failure rate was around 99%. This makes stabilizing the pendulum without eligibility very time consuming when tuning parameters since almost no combination of values work.

For the quantized IKRLS-CMACs to perform well, the linear sensitivity function had to be used. The binary sensitivity function was simply not good enough as its resolution and approximation to the full overlay is too poor at the low generalization parameter value used to maintain computational efficiency. Credit assignment was able to further improve the errors significantly. The CA2LE-CMAC was able to also reduce the error, but not as well as the CALE-CMAC. It was also found the quantized CMAC computation times are not significantly affected by the size of the eligibility decay time $\psi$, and that computation times are only slightly increased by use of eligibility.

The continuous IKRLS-CMACs with vector eligibility also stabilized the inverted pendulum well, but were significantly computationally slower when compared with the quantized CMACs when the decay time was set to $\psi = 1000$. However, by reducing the eligibility decay times to $\psi = 100$ the continuous CMAC was able to achieve computation times closer to that of the quantized CMACs without increasing the error significantly. However, although the computation times were reduced, the actual computational speed of the continuous IKRLS-CMACs was still slower than the CALEh25-CMAC. It was also discussed how the sample rate affects the decay time choice, and that continuous CMACs sampled at a high sample rate will need a larger eligibility decay time, causing greater computational burden. The results also showed that using a decay time much smaller than the eligibility profile response decay time did not affect the CMAC error significantly. Additionally, it was shown that the third-order kernel outperformed the zero-order kernel, but required a small increase in computation time.

In conclusion the results showed that the CMACs developed in this work outperform the other third party CMACs tested in this experiment. The quantized IKRLS-CMACs operate at computational speeds that are slower, but still competitive with the LMS based FOX-CMAC. The quantized CMAC has an advantage over the continuous CMAC in that it can use credit assignment. However, the continuous CMAC can make up for the lack of the credit assignment enhancement by offering more accurate modeling due to its use of the full overlay and infinite quantizing resolution. The CMACs with the best error computation time trade-off were the CALE-CMAC and CALEh25-CMAC, which had very low errors, and also fast computation speeds.
Figure 10.14: Results of Eligibility Profiles Tested - Pole angle results of the poorly chosen eligibility profiles. **Key:** Poorly matched eligibility response one (---); Poorly matched eligibility response two (----); Good eligibility response (···).
10. MOTION CONTROL TEST - INVERTED PENDULUM
Motion Control Test - Gantry Crane

A gantry crane is a type of crane that consists of a motorized cart which moves along a railing, and moves a hanging load which is attached to the cart by a cable. A problem with gantry cranes is that they are prone to causing an oscillating swing motion in the load. The goal of this experiment is to design a controller that minimizes this swing. This type of experiment has been previously performed in [110] with a Lyapunov control based Fuzzy CMAC and in [6] with a FOX-CMAC. Here, an experiment similar to the FOX-CMAC one is performed with the IKRLS-CMACs, and also some other well established CMACs.

In this experiment MATLAB simmechanics is used to model and simulate the crane system and the block diagram of the crane is shown in Fig. 11.1. The simulated crane mass is 5 kg, the load mass is 1 kg with a cable length of 1 m and negligible cable mass. The crane system uses a ‘Kinetic Damping Actuator’ on the cart to simulate frictional drag, and a ‘Link Springiness’ block on the cable to simulate cable springiness. The control system used is shown in Fig. 11.2. The simulink simulation used a fixed timestep of 0.1 seconds and ODE3 as the solver.

The CMACs tested all used an input of $\mathbf{u} = \begin{bmatrix} p_c & p_l & \frac{dp_c}{dt} & \frac{dp_l}{dt} \end{bmatrix}$, where $p_c$ is the gantry crane cart position and $p_l$ is the gantry crane load position. The input value limits for quantization and normalization are bounded between $i_{\text{max}} = [1.2 \ 1.2 \ 0.7 \ 0.7]$ and $i_{\text{min}} = [-1.2 \ -1.2 \ -0.7 \ -0.7]$. All quantized CMACs used the uniform overlay and also the approximation to $\mathbf{P}$ described in Section 3.4.1.

The test requires the crane to move a load back and forth with a square wave trajectory. The square wave reference signal has an amplitude of $\pm 1$ and has a frequency...
of 0.008 Hz, completing a trip once every 125 timesteps or 12.5 seconds.

The feedback controller is a simple proportional based system that very roughly tries to move the load to a desired location whilst avoiding large oscillations in the load. Its equation is given by (11.1). It tries to move the cart towards the reference position, but if the load moves too far from the cart it slows down the cart speed to prevent an overly large gap between cart and load, thus reducing swing oscillation.

\[ y = -3(p_c - \text{ref}) + 2(p_l - \text{ref}) \quad (11.1) \]

For eligibility, the system impulse response was measured by setting \( \text{ref} = 0 \) and then applying an impulse to the load and measuring the response. A second order eligibility profile approximation was fitted to the response through trial and error. Its parameters are given by \( p_a = 0.15, \ p_b = 0.5, \ p_c = 1, \ p_h = 0.1 \) with a decay period of 250 timesteps. The impulse response and second order approximation is shown in Fig. 11.3. Also, the eligibility matrices are reset every time the reference is changed as discussed in Chapter 6.

### 11.1 Results

In this section the improvement that is brought to the quantized and continuous IKRLS-CMACs by each enhancement is tested. The CMACs developed in this work are also tested against other CMAC variants such as the FOX-CMAC, the SOCM and the LMS-CMAC.

As in the previous chapter involving the inverted pendulum, the tests are performed over a wide variety of parameters. For this experiment, the total absolute error (TAE) between the desired square wave reference load position and the actual load position is measured for 2000 timesteps or 200 seconds (3.33 minutes) of simulated time.

The tests are carried out over the ranges \([0.2 : 0.02 : 0.04], [0.3 : 0.1 : 0.9]\) and \([0.05 : 0.05 : 0.6]\) for the generalization parameter \( \phi \), sparsification parameter \( \nu \) and learning rate \( g_e \) respectively for all IKRLS-CMAC experiments. The FOX-CMAC was tested over the learning rate range \([0.01 : 0.01 : 0.2]\), LMS-CMAC over the learning rate range \([0.0001 : 0.0002 : 0.01]\) and the SOCM over the range \([0.00001 : 0.00003 : 0.001]\). These learning rate ranges and decimations were chosen in the same way that was used in the Inverted Pendulum experiment from the previous chapter.
Figure 11.1: Simmechanics Gantry Crane - Simmechanics Crane Implementation.
Figure 11.2: Simmechanics Crane - Simmechanics Crane control system implementation.

Figure 11.3: Gantry Crane Impulse Response - The first 500 timesteps at a sample rate of 0.1 seconds per timestep for the response of the load position when an impulse is applied to the load. **Key:** Impulse response (—); The approximated second order eligibility profile (———).
All CMACs used the second order eligibility decay described previously, and a decay time of $\psi = 250$ timesteps. The KRLS weight decay parameter $\tau$ was set to $\tau = \frac{1}{50}$ as usual.

As with the inverted pendulum and regression tests the minimum errors for each set of $\phi$ was taken to decouple the results from the learning rate and sparsification parameter $\nu$ selections.

### 11.1.1 Comparison Error Results

In Fig. 11.4 the ablative and comparison test results are shown as a box plot. The results show that, just as with the inverted pendulum, the quantized binary sensitivity based CMACs performed poorly in terms of their median TAE results when compared to the linear sensitivity based CMACs. In this test, credit assignment produced a much larger improvement in the median compared to the inverted pendulum results. The CA2LE method performed better than the LE-CMAC, but not as well as the CALE-CMAC. The CA2LE also had a larger spread when compared to the CALE-CMAC.

The CALEh29-CMAC, a CALE-CMAC with $h = 29$, was the best performer out of all the CMACs tested.

The C0E-CMAC performed similarly to the CA2LE-CMAC, but with a much reduced spread. The C3E-CMAC was the best performing CMAC without credit assignment.

The FOX-CMAC performed poorly when compared to all the IKRLS based CMACs, but was still able to provide a satisfactory solution in most cases.

The LMS-CMAC is not shown in the graph as it performed very poorly, and was unable to stabilize the swing of the load. The results were even worse than the feedback controller by itself. The SOCM is also not shown due to its poor performance. It performed slightly better than the feedback controller by itself, but was still extremely poor in comparison to the eligibility based CMACs.

### 11.1.2 Average Computation Time Results

Here the average computation time results over multiple training runs are compared in order to compare the computational speed differences between the various CMAC implementations.
In Fig. 11.5 the computation time results per iteration for the different tests on the crane control problem are shown. Additionally, Fig. 11.6 shows the number of weights used for each CMAC test. The computation time and number of weights results are the results from the CMACs that obtained the minimum error results gathered in the error comparison section.

The results show that with credit assignment turned off the binary and linear quantized IKRLS-CMACs have similar computation times. However, like with the inverted pendulum, the binary sensitivity CMAC used more weights than the linear sensitivity CMAC, which contributed to the increase in its computation time.

Credit assignment increases the weight usage for both the binary and linear CMACs and also increased their average computation times per iteration. The CALEh29-CMACs computation time was increased significantly from the CALE-CMACs. The CA2LE-CMAC had a computation time that was in between the LE and CALE-CMAC, as the CALE and LE-CMACs used a similar number of weights since they share the same sparsification algorithm.

The C0E-CMAC had computation times similar to the CALEh29, but with more spread. Despite using less weights than the zero order kernel, the C3E-CMAC had a longer average computation time due to its more complex kernel generation algorithm.
As expected the FOX-CMAC was very fast being an LMS based solution, however most of the quantized IKRLS-CMACs remain fairly competitive in terms of computation times, with the CALE-CMAC being about four times slower than the FOX-CMAC on average. The LMS-CMAC was the fastest as it is the simplest. The SOCM was very slow as it is a kernel based solution and tended to accept a lot of weights into the dictionary with the sparsification method it used.

### 11.1.3 Actual Computation Time Results

The actual computation time results for the CMAC variants that performed well are shown in Fig. 11.7 for the case where a weight is not added into the dictionary, and in Fig. 11.8 for the case when a weight is added into the dictionary. Results were collected up till a maximum of 150 weights were added into the dictionary. The results show that more computational time is required for the situation when a weight is added into the dictionary.

In terms of actual computational speed rankings, the LE-CMAC is the fastest followed by the CALE, CALEh29 and finally the C3E-CMAC. The CA2LE-CMAC runs at about the same speed as the LE-CMAC when a weight is not added, but is slower than the CALE-CMAC when a weight is added, as it must do two inverse kernel updates.

As expected, the C3E-CMAC was the most computationally intensive CMAC tested.
Figure 11.6: Gantry Crane Weight Usage - Box plots showing weight usage comparisons for the crane reference tracking problem.

Figure 11.7: Gantry Crane Actual Computation Times (Weight Not Added) - Actual computation times for when a weight is not added to the dictionary. Key: CALEh29-CMAC (—); LE-CMAC (—); CALE-CMAC (— —); C3E-CMAC (— — —); CA2LE-CMAC (⋯).
11.1 Results

![Graph showing computation times vs. number of weights]

Figure 11.8: Gantry Crane Actual Computation Times (Weight Added) - Actual computation times when a weight is added to the dictionary. **Key:** CALEh29-CMAC (—); LE-CMAC (—); CALE-CMAC (——); C3E-CMAC (-----); CA2LE-CMAC (••). Notice in the graph how the computation times slowly rise and then fall instantly again. The increase in computation time is due to the eligibility matrix growing in size up to a length of $\psi$. The sudden decrease is due to all eligibility matrices being emptied when the reference value changes, as discussed in Section 6.3.4.

On average, the linear sensitivity based CMACs used between 50 and 80 weights for the best parameter choices used in the ablative tests above. In Table 11.1 the computation times and resulting maximum sample rates for when an input is added to the dictionary are compared for 80 dictionary weights. Note that the computational time value for the C3E-CMAC is interpolated between the peaks of the graph. This is because the peak positions may change depending on the timing of the eligibility profile reset. The table shows that maximum sampling rates are fast enough for many real time operations. Again, note that as discussed in the previous chapter, the continuous CMAC computation speed will be dependent on the sample rate used, where in this example a sample rate of 10 Hz was used.
11. MOTION CONTROL TEST - GANTRY CRANE

Table 11.1: Maximum Actual Computation Times for the Gantry Crane Experiment when \( n_s = 80 \)

<table>
<thead>
<tr>
<th>CMAC Type</th>
<th>Computation Time (ms)</th>
<th>Max Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE</td>
<td>1</td>
<td>1000 Hz</td>
</tr>
<tr>
<td>CALE</td>
<td>1.1</td>
<td>910 Hz</td>
</tr>
<tr>
<td>CA2LE</td>
<td>1.4</td>
<td>714 Hz</td>
</tr>
<tr>
<td>CALEh29</td>
<td>1.5</td>
<td>667 Hz</td>
</tr>
<tr>
<td>C3E</td>
<td>3</td>
<td>333 Hz</td>
</tr>
</tbody>
</table>

11.2 Conclusion

In this chapter the various CMAC variants developed in this thesis and from third parties were tested on an online simulated gantry crane stabilization problem. The CMACs from this thesis that were tested were the quantized and continuous IKRLS-CMACs. The third party CMACs tested were the LMS, SOCM and FOX CMACs.

The results obtained were similar to those obtained in the inverted pendulum tests, where the binary sensitivity based IKRLS-CMACs performed poorly in comparison to the linear sensitivity based IKRLS-CMACs. However, in this experiment credit assignment demonstrated a greater improvement than in the inverted pendulum tests. The secondary credit assignment method performed once again only slightly poorer compared to the normal credit assignment method.

The C0E-CMAC performed better than the LE-CMAC, but not better than either the CALE-CMAC or the CA2LE-CMAC. This shows that in this experiment, credit assignment gave a large improvement to the error results. The CALEh29-CMAC was the best overall performer.

The computation times for the quantized linear IKRLS-CMACs were fast enough for real time use, and did not differ too much from the FOX computation time.
Motion Control Test - Position Controlled Inverted Pendulum

For a harder test, the goal of the inverted pendulum experiment was modified to simultaneously control the inverted pole angle, and make the cart follow a square wave reference pattern. The same inverted pole system seen in Chapter 10 is used. However, this test used a simulation timestep of 0.03 seconds to speed up simulation, as the simulated time must be longer to capture enough data.

First a standard state-feedback control system was roughly designed and tuned to a point at which the cart would move towards the reference position, but eventually diverge and fail. The state feedback controller is described by (12.1) where ref is a discontinuous square wave that switches between values of 0.5 and -0.5 with a frequency of 0.01Hz.

\[
y = -130(\theta - \text{ref}) + 8p_c - 50\frac{d\theta}{dt} + 2\frac{dp_c}{dt}
\]

(12.1)

This controller attempts to simultaneously balance the pole and push the cart towards the reference by causing the pole to be always falling towards the cart reference.

The inputs to the CMACs tested are given by \( \mathbf{u} = \begin{bmatrix} \theta & \frac{d\theta}{dt} & p_c - \text{ref} & \frac{dp_c}{dt} \end{bmatrix} \). Note that in this test the eligibility matrix and decay vector is reset every time the reference value switches, as is discussed in Chapter 6.

In these tests the quantization/normalization limits are set as in the following, \( i_{\text{max}} = [0.2 \ 1 \ 0.3 \ 1] \) and \( i_{\text{min}} = [-0.2 \ -1 \ -0.3 \ -1] \). The error to the CMAC is given by \( e = p_c - \text{ref} \). Note that the pole angle error \( \theta \), is not specified to the CMAC error function. This is not a problem as the cart cannot attain a stable reference position without the pole angle also being simultaneously well controlled. This is the same approach used in a similar experiment performed in [6].
12. MOTION CONTROL TEST - POSITION CONTROLLED INVERTED PENDULUM

\[ \begin{bmatrix} 1 & 0.0681 & 0.0115 \\ -0.2668 & 0.7445 & 0.0175 \\ 0.5276 & 0.6628 & 0.9809 \end{bmatrix} \]

(12.2)

\[ b = \begin{bmatrix} 0 \\ 0 \\ 0.4993 \end{bmatrix} \]

(12.3)

\[ c = [0 \ 1 \ 0] \]

(12.4)

**Figure 12.1:** Simmechanics Position Controlled Inverted Pendulum - Simmechanics position controlled inverted pendulum control system implementation.

In Fig. 12.1 the control system used in this experiment is shown. The cart position reference is set by the square wave generator, and is used as the reference input to the feedback and CMAC controllers. The impulse response of the cart position \( p_c \), was measured by setting \( \text{ref} = 0 \) and applying a small impulse to the pole pin joint (with feedback controller connected), and then measuring the cart position sensor. The impulse response of the cart position is shown in Fig. 12.2 as the divergent solid line. It was found that the impulse response was difficult to model with a second order eligibility profile as the profile model peaks would not line up with the real data. Thus instead, a third order eligibility profile was evolved using the genetic algorithm method described in Section 6.6. Since the impulse response is also divergent, the method shown in Section 6.6 for divergent impulse responses was used to first force the data to converge to zero. The evolved eligibility profile is shown in Fig. 12.2 as a dashed line. It is given by the matrices (12.2), (12.3) and (12.4) and has a settling time of approximately 500 timesteps.
12.1 Results

In these tests the CMACs used a decay threshold of $\psi = 300$ and a generalization parameter of $h = 15$ unless otherwise specified. The same testing method utilized in the previous test chapters was used. The generalization parameter was $\phi$ was iterated over $[0.1 : 0.02 : 0.9]$, sparsification parameter $\nu$ was iterated over $[0.4 : 0.1 : 0.9]$ and the learning rate $\alpha$ was iterated over $[0.05 : 0.02 : 0.27]$.

12.1.1 Comparison Test Results

In Fig. 12.3 the comparison test results are shown as a box plot. Note that in this experiment, unlike the previous experiments, only the results for the IKRLS-CMAC variants developed in this work are shown, as the FOX, LMS and SOCM CMACs were unable to stabilize the pendulum in this test at all for any tested parameter choice. This is probably because there is no initial gain setting available for these CMACs. Additionally, the binary sensitivity results for the quantized IKRLS-CMACs are also not shown as those CMACs tended to accumulate a large number of weights, causing the experiment to run too slowly to gather results for in a reasonable period of time.

The results show that credit assignment in the CALE-CMAC significantly reduced the error when compared to the LE-CMAC. The secondary credit assignment method
12. MOTION CONTROL TEST - POSITION CONTROLLED INVERTED PENDULUM

![Boxplot of Total Absolute Error for Different Methods]

Figure 12.3: Position Controlled Inverted Pendulum Error Results. - Errors showing how well each CMAC variant was able to follow the reference trajectory where the errors used are the best results for each value of $\phi$ tested.

in the CA2LE-CMAC also performed better than no credit assignment, but was not as good as the CALE-CMAC.

Increasing the generalization parameter to $h = 29$ as in the CALEh29-CMAC also reduced the error further.

The C0E-CMAC performed better than the LE-CMAC, but not better than the CALE-CMAC. The C3E-CMAC however performed slightly poorer when compared to the C0E-CMAC. It is expected that there will be some cases in which the binary sensitivity function can outperform the third order b-spline in the continuous CMACs.

In Fig. 12.4 the cart position graph for the parameters that gave the lowest total absolute error for the CALEh29-CMAC is shown. These parameters where $h = 29$, $\phi = 0.55756$, $\nu = 0.5$ and $g_e = 0.25$. This figure shows how well the CALEh29-CMAC controls the cart. Overshoot could be further reduced by utilizing output derivative limiting, however that is not tested here.

12.1.2 Average Computation Time Results

In Fig. 12.5 the overall average computation time per iteration results for the tests performed are shown. From these tested CMAC variants it can be seen that the CA2LE-CMAC and LE-CMAC were the fastest. The primary credit assignment method used
in the CALE-CMAC increased computation times over the LE-CMAC as expected. Increasing the generalization parameter from $h = 15$ to $h = 29$ in the CALEh29-CMAC also further increased the computation time as expected.

The continuous CMACs were again the slowest variants being significantly slower compared to the quantized CMACs. As the continuous CMACs did not reach an error as low as the CALEh29-CMAC, there seems to be no reason to choose them over the CALE or CALEh29-CMACs for this problem.

In Fig. 12.6 the number of weights used by the CMAC variants is shown. All the linear sensitivity based CMACs used a similar number of weights. The C0E-CMAC on the other hand, used significantly more weights in comparison, perhaps indicating that a smoother control scheme was used with the linear sensitivity function.

### 12.1.3 Actual Computation Times

In Fig. 12.7 the actual computation times for when a weight is not added into the dictionary are plotted against the number of weights used in the CMAC.

As expected, the LE-CMAC and CA2LE-CMACs are the fastest performers. Credit assignment increases the computation times, and so does increasing the generalization
Figure 12.5: Position Controlled IP Average Computation Times - Average computation times per iteration for the best error value parameters.

Figure 12.6: Position Controlled IP Number of Weights - Number of weights used by the CMAC tested with the best error value parameters used.
12.2 IKRLS-CMAC Compared against Other Methods on Motion Control Learning

Table 12.1: Actual Computation Times when $n_o = 127$

<table>
<thead>
<tr>
<th>CMAC Type</th>
<th>Computation Time (ms)</th>
<th>Max Sample Rate (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE</td>
<td>1.6</td>
<td>625</td>
</tr>
<tr>
<td>CALE</td>
<td>1.7</td>
<td>588</td>
</tr>
<tr>
<td>CA2LE</td>
<td>2.5</td>
<td>400</td>
</tr>
<tr>
<td>CALEh29</td>
<td>2.5</td>
<td>400</td>
</tr>
<tr>
<td>C3E</td>
<td>5</td>
<td>200</td>
</tr>
</tbody>
</table>

parameter as expected. The C3E-CMAC is the slowest CMAC. It is expected to be slow as the decay parameter is set quite large at $\psi = 300$. As in the previous tests with the gantry crane, the spikes and dips in computation times occur when the reference changes and the eligibility data is reset.

In Fig. 12.8 the actual computation times for the case when a weight is added into the dictionary are shown. Adding a weight into the dictionary is a more computationally intensive process as can be seen by the increase in computation times when compared to Fig. 12.7. The CA2LE-CMAC is much slower when adding a weight as it must perform two kernel matrix updates, instead of only one. The speed rankings of the other CMAC variants remain the same as in Fig. 12.7.

The CMACs shown in these graphs used between 108 - 127 weights for their best parameter selections. In Table. 12.1 the computation time for 127 weights is shown, and the corresponding maximum sample rates are calculated. These results show the CMACs can operate at sample rates that should be sufficient for many online applications.

12.2 IKRLS-CMAC Compared against Other Methods on Motion Control Learning

The IKRLS-CMAC is one of the few methods that can control a system from scratch online. One alternative might be adaptive learning through fuzzy networks. In [111] Yamakawa et al. used a fuzzy rule-based system to control an inverted pendulum. Yamakawa's system however had no adaptive means and required humans to input the fuzzy rules manually. Later the DENFIS system was created in [108]. In some ways the DENFIS network used in an online motion control problem can be thought of as similar to an IKRLS-CMAC in a feedback error control learning system. Where in the DENFIS network the role of the initial PID controller is fulfilled by the act of manually
Figure 12.7: Position Controlled Inverted Pendulum Actual Computation Times (Weight Not Added) - Actual computation times when no weight is added to the dictionary. **Key:** CALEh29-CMAC (---); LE-CMAC (--); CALE-CMAC (---); C3E-CMAC (---); CA2LE-CMAC (···).

Figure 12.8: Position Controlled Inverted Pendulum Actual Computation Times (Weight Added) - Actual computation times when a weight is added to the dictionary. **Key:** CALEh29-CMAC (---); LE-CMAC (--); CALE-CMAC (---); C3E-CMAC (---); CA2LE-CMAC (···).
entering fuzzy starting rules. DENFIS has not yet been tested in online motion control with feedback error learning but we would expect it to have similar performance to an improved CMAC without the eligibility improvement.

Many other more common neural networks such as MLPs and RBFs have been tested in feedback error learning inverted pendulum control but most require several training runs and potential failures before they can learn the control function. Whereas in comparison the IKRLS-CMAC is able to control the system instantly. Often MLPs and RBFs are successfully used in other control schemes involving linearized models or learned (identified) models of the system. These methods are usually quite different to feedback error learning as they require some sort of initial modeling or learning which is not required by the IKRLS-CMAC in feedback error control learning.

Recently there has been much work on providing a guarantee of stability through Lyapunov control theory [107] when combining neural controllers with conventional control schemes such as sliding mode control. The main disadvantage to the IKRLS-CMAC in feedback error control learning mode is that no such similar guarantee of stability can be made. However, these guarantees in other works come at the cost of using other more complex control schemes which require linearized models, initial modeling or system identification.

12.3 Conclusion

In this section a position controlled inverted pendulum was tested with the CMACs developed in this work and some already well known CMACs.

The results showed that the well known CMACs could not even stabilize the system, whereas the IKRLS-CMACs developed in this work could, due to the use of the initial gain parameter to prevent initial divergence, and vector eligibility. The results also showed that credit assignment significantly reduced the test error of the system. The continuous CMACs were not able to reach an error as low as the quantized credit assigned CMACs as they do not have credit assignment. Additionally, it was shown that the continuous CMACs had computation times that were much slower than the quantized CMACs. It is recommended that a CALE based CMAC be used with a reasonably sized generalization parameter, as that configuration produced the lowest error and was still computationally efficient.
12. MOTION CONTROL TEST - POSITION CONTROLLED INVERTED PENDULUM
Conclusions and Future Work

13.1 Main Thesis Contributions

1. **Goal** - Implement a Recursive Least Squares CMAC. **Completed** - A more efficient recursive least squares (RLS) CMAC based on the IQR-RLS algorithm was created. Also, a method for implementing regularization/credit assignment into the IQR-RLS-CMAC was derived.

2. **Goal** - Implement a Kernel RLS-CMAC. **Completed** - A kernelized version of the RLS-CMAC was created, and called the KRLS-CMAC.

3. **Goal** - Implement Higher Order Sensitivity Functions into the KRLS-CMAC. **Completed** - Higher order sensitivity functions were implemented into the KRLS-CMAC via a modified b-spline algorithm.

4. **Goal** - Investigate Improved Overlays. **Completed** - The ability of the uniform overlay to approximate the full overlay was investigated and useful results that can help choose the generalization parameter were obtained. Tests comparing two methods of combining multidimensional kernel functions were undertaken and one method was found as superior.

5. **Goal** - Implement Regularization/Credit Assignment in the KRLS-CMAC. **Completed** - Two iterative methods for using credit assignment in the KRLS-CMAC were derived.

6. **Goal** - Implement Vector Eligibility into the KRLS-CMAC. **Completed** - A method for using vector eligibility in the KRLS-CMAC was created. Also a method based on genetic algorithms was developed for finding eligibility profiles.
7. **Goal** - Find out if it is possible to use all the improvements together. **Completed** - The KRLS-CMAC was combined together with higher order sensitivity functions, credit assignment and vector eligibility. This combined algorithm is called the Improved KRLS-CMAC (IKRLS-CMAC).

8. **Goal** - Find out if a continuous non-quantized CMAC with the above improvements is possible. **Completed** - A continuous version of the IKRLS-CMAC was created, and higher order sensitivity functions and continuous vector eligibility was implemented into it. Additionally, a working, but very computationally intensive credit assignment method for the continuous IKRLS-CMAC was created.

### 13.2 Suggested Future Work

1. The most obvious continuation of this work is an application of the IKRLS-CMAC to a physical motion control system rather than a simulation.

2. The IKRLS-CMAC MATLAB code should be ported to C/C++ or another efficient language for improved computational efficiency; MATLAB is known to be computationally inefficient for some code.

3. The IKRLS-CMAC could be implemented into hardware for much improved computational performance. This would pose several problems, the main issue being that a growing memory structure would be required.

4. Further investigation into credit assignment for the continuous IKRLS-CMAC needs to be undertaken. An iterative update to the inverse kernel matrix and kernel weight vector needs to be derived. Is there perhaps an alternative and more computationally efficient method to avoiding learning interference for the continuous version of the KRLS-CMAC?

5. Alternative overlay arrangements should be investigated which favor reducing the difference between the reduced overlay kernel response and the full overlay kernel response. Is the uniform overlay optimal in this sense? A two constraint genetic algorithm could be used to find a Pareto minimum between kernel error and number of layers $m$.

6. An investigation into using the IKRLS-CMAC in alternative control schemes, such as when combined with sliding mode and Lyapunov control theory as in other works involving the CMAC should be undertaken.
7. If a KRLS algorithm is not required, vector eligibility could be implemented in the KLMS-CMAC in the same way that it is implemented in the quantized IKRLS-CMAC.

8. An investigation into hierarchical IKRLS-CMACs to reduce computation times for very large problems should be done.

9. A method for applying self organization to the dictionary in the IKRLS-CMAC should be investigated. This would likely also involve a method for pruning the dictionary.

10. Implementation of the extended KRLS algorithm should be investigated.

13.3 Conclusions

In this thesis a new CMAC online neural network controller was developed which unites several key enhancements found in the CMAC literature into a single CMAC algorithm called the IKRLS-CMAC. Wherever a CMAC is currently used in a feedback error learning situation, the IKRLS-CMAC should be considered as a potential replacement that will learn faster, and also learn a more accurate solution. Whilst the IKRLS-CMAC is not on the same level of computational simplicity as the standard Albus-CMAC, it still retains competitive computational efficiency and it is more than capable of real time operations on a modern mid range Intel i5 750 CPU.

The first CMAC enhancement applied was the use of the RLS algorithm as the learning method, rather than the LMS algorithm. This provided the advantage of convergence within a single cycle for a prepared training set. The RLS algorithm also worked well with online motion control learning problems by effectively modifying the learning rates automatically. The use of RLS however introduced a major flaw. The computation time of the RLS algorithm was vastly increased to the point of unusability in some cases due to the CMAC architecture. Some improvements to the RLS algorithm involving IQR decomposition were applied, but although computational speed was improved, the computational complexity remained the same. It seemed the RLS algorithm was not a good match for the CMAC.

The second enhancement applied was the kernelization of the RLS-CMAC through use of the KRLS algorithm. This enhancement was applied to overcome the computational performance flaw introduced by RLS. The KRLS algorithm was able to overcome the flaw by transforming the computational complexity to be dependent on the size of
13. CONCLUSIONS AND FUTURE WORK

a sparse dictionary of input data, rather than the size of the CMAC association vector, where the size of the dictionary is much less than the size of the association vector. The KRLS algorithm also brought sparsification with it. Sparsification of the kernel dictionary helps to prevent over training and thus improves the CMACs generalization ability.

The third enhancement applied was the use of higher order sensitivity functions. The standard CMAC uses a binary sensitivity function which in turn causes the CMAC output to be discontinuous and stair like. Using a linear sensitivity function allows the CMAC output to be smooth and continuous. It also improves the CMAC with uniform overlay's kernel approximation to the full overlay. Two-norm normalization of the association vector was also applied which also improved the kernel approximation.

The fourth enhancement applied was credit assignment. It was discussed how credit assignment and regularization are essentially the same method. This enhancement reduced the learning interference effect that occurs in CMAC based neural networks. This was applied to the KRLS-CMAC and required derivation of some extra calculations that are used to update the inverse kernel $K^{-1}$ matrix and the kernel space weight vector $\alpha$.

Finally, the fifth improvement that was applied was vector eligibility. Vector eligibility vastly improves the online motion control learning performance of the IKRLS-CMAC. With vector eligibility the IKRLS-CMAC is able to control motion control problems that it could not previously in a feedback error learning configuration.

A continuous IKRLS-CMAC that does not need to calculate the association vector was also derived. This continuous IKRLS-CMAC implicitly uses the full overlay. The continuous IKRLS-CMAC is closer to classic kernel machines, but retains most of the same properties as the CMAC. It was found that applying credit assignment to the continuous IKRLS-CMAC was difficult, but a relatively computationally intensive method involving combinatorics showed that it is possible. Vector eligibility was also implemented into this version of the IKRLS-CMAC. However, it was found that in the continuous IKRLS-CMAC, vector eligibility is potentially computationally slower than when used in the quantized IKRLS-CMAC. It's computational complexity is dependent on the eligibility decay time, and if it is large, it may be too computationally inefficient to use online.

The developed IKRLS-CMACs were successfully tested on multidimensional regression problems, a system identification problem and several simulated online motion
control problems including an inverted pendulum, gantry crane and position controlled inverted pendulum. Tests were carried out to determine the effect that each CMAC enhancement had on the system. The results showed that each enhancement improved upon the error results, and that the enhancements were able to work harmoniously together. Additionally, the IKRLS-CMAC was compared against the FOX-CMAC and other already well known CMAC variants, and was shown to be superior in each tests results.

In addition to the combination of CMAC enhancements, several theoretical insights into the operation of the CMAC overlays were gathered. In [9] it was suggested that the minimum function be used to combine multidimensional sensitivity functions in the kernel calculation. In this thesis these claims were investigated. It was found that in some cases when the generalization parameter was small and the CMAC dimensionality was greater than two, the minimum function was slightly better. However, for values of $h$ larger than a certain value depending on the CMAC dimensionality, the radial function was better. Additionally, it was found that normalizing the association vector by the two-norm of the association vector significantly improved the approximation ability of the uniform overlay, and made it such that the radial function was always the better choice.

Furthermore, the effect of the generalization parameter value when using reduced overlays (such as the uniform overlay) was investigated. The results showed that larger generalization parameter values created kernel responses that were a better approximation of the full overlay kernel response. This investigation has created a better method for selecting the generalization parameter which does not rely solely on trial and error.

In conclusion, this thesis has successfully presented a new form of CMAC that outperforms other CMAC implementations for online motion control problems, when used in a feed back error learning situation. It has shown that it is possible to combine the chosen CMAC enhancements into a single algorithm.
13. CONCLUSIONS AND FUTURE WORK
Appendix A

Full Overlay Base Conversion Algorithm

The full overlay for the CMAC can be easily generated on the fly with a simple algorithm that involves converting number bases. The rows of the full overlay displacement matrix can be obtained simply by converting the values 0 to $m - 1$ into base $h$.

For example, to calculate the full overlay displacement matrix for $h = 3$ for a two dimensional CMAC, first calculate the number of layers required with $m = h^2 = 3^2 = 9$. This means that the displacement matrix will be of dimension $m \times n_d$ which in this example is $9 \times 2$.

Now, all that is left to do is simply convert the values of 0 to $m - 1$ into base $h$, where each row in the displacement array will be populated by the results. The complete table for this example is shown in Table A.1. Note that if a value below $n_d$ digits is generated, such as with 0, 1, 2 in this example, filler zeros must be added to make the number $n_d$ digits long.

The algorithm for generating the full overlay for any arbitrary $h$ is shown in Algorithm 23, where the ConvertBase function is shown in Algorithm 24.

---

**Algorithm 23** Calculate the Full Overlay Displacement Matrix

1: procedure GETFULLOVERLAYDISPLACEMENTMATRIX($h, n_d$)
2: \hspace{1cm} $m = h^{n_d}$
3: \hspace{1cm} for $i \leftarrow 1 : m$ do
4: \hspace{2cm} $D_{i,1:n_d} \leftarrow $ ConvertBase($i - 1, h, n_d$)
5: \hspace{1cm} end for
6: end procedure
A. FULL OVERLAY BASE CONVERSION ALGORITHM

Table A.1: Full Overlay via Base Conversion for $h = 3$.

<table>
<thead>
<tr>
<th>Row</th>
<th>Base 3 Conversion</th>
<th>$D_1$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Algorithm 24 Base Conversion Algorithm

1: procedure CONVERTBASE(base10Num, desiredBase, $n_d$)
2:     temp = base10Num
3:     $d_{1:n_d} = 0$
4:     $c = 1$
5:     while (temp > desiredBase) do
6:         $d_c = \text{mod}(\text{temp}, \text{desiredBase})$
7:         temp = $\left\lfloor \frac{\text{temp}}{\text{desiredBase}} \right\rfloor$
8:         $c = c + 1$
9:     end while
10:    $d_c = \text{mod}(\text{temp}, \text{desiredBase})$
11:    $c = c + 1$
12:    division = $\left\lfloor \frac{\text{temp}}{\text{desiredBase}} \right\rfloor$
13:    if division $\neq 0$ then
14:        $d_c = \text{division}$
15:    end if
16: return d
17: end procedure
Appendix B

Introduction to Genetic Algorithms

In this appendix, a very brief introduction to genetic algorithms will be provided to introduce those people that have no knowledge in this area of computational intelligence.

The Genetic Algorithm (GA) was first described in 1975 by John Holland [112]. GAs are a class of evolutionary algorithms that are inspired by the ideas of evolutionary natural selection. In natural selection, the fittest organisms in a population that exhibit the best survival traits are the ones that are most likely survive to mate and produce offspring. Over many generations of the population of organisms, the best survival traits will dominate.

B.1 Genetic Algorithm Terminology

B.1.1 Phenotype

A phenotype is the description of an organisms traits. For example a phenotype may describe a human with blue eyes and brown hair.

In computational evolution the phenotype is a candidate solution. The candidate solution will contain a number of genes, where each gene describes one part of the solution. For example, a phenotype may be a bit string, where each bit is a gene which describes whether a switch should be on or off.

B.1.2 Fitness

The fitness is a measure of a phenotypes success at some task. In nature, this success is usually defined by an organisms ability to survive long enough to mate.
B. INTRODUCTION TO GENETIC ALGORITHMS

In computational evolution, this success may be defined. For example, success may mean a more optimal solution to a particular problem. More optimal solutions will have greater fitness.

B.1.3 Evaluation

Evaluation is the act of evaluating an organisms fitness through some sort of test. For example a test might evaluate the number of timesteps a system has been stable for under an evolved controller.

B.1.4 Selection

In nature usually the fittest organisms are the ones that get to reproduce, as the unfit organisms will either die prematurely, or will be unattractive to mates.

In computational evolution selection consists of selecting the fittest organisms through various means such as probabilistic methods, or simply by ranking the organisms in terms of fitness and choosing the best.

B.1.5 Reproduction/Crossover

In nature most successful organisms reproduce sexually. Sexual reproduction is when genes from two parents are combined together in some way to form a new chromosome that describes the child organism. As the child organism contains genes from both parents, it will exhibit traits from both parents.

In computational evolution, reproduction is called crossover and is used to converge the population towards a solution. It is usually applied by taking half the genes from two parents and combining the two halves in some way to create a child. The phenotypes selected from the selection process will perform crossover and pass their fit genes to the next generation.

B.1.6 Mutation

In nature mutation occurs when errors occur during copying of a gene from parent to child.

In computational evolution mutation is purposely implemented usually by randomly altering a gene after crossover. Mutation is important as it helps diversify the solutions, and helps avoid the algorithm getting stuck in a local minima.
B.2 Genetic Algorithm Flowchart

In Fig. B.1 a flow chart describing a typical genetic algorithm is shown.

First the population is initialized. This is where a large population of phenotypes or candidate solutions are randomly generated. Next the fitness of this population is tested. The fittest solutions are selected in some way and then crossover and mutation is performed on these fit solutions in order to create a new child generation of candidate solutions. The working population is now replaced with the child population and the process resumes from the fitness test.

B.3 Advanced Genetic Algorithms

The standard genetic algorithm described above does not work that well in most cases. Various more advanced genetic algorithms exist such as Enforced Subpopulations (ESP) [113] which maintains multiple individual populations of candidate solutions that evolve in isolation to one another. Only sometimes does inter-population mating occur. The MATLAB genetic algorithm used in this work implements this concept and is used to help evolve the eligibility profile.

Other genetic algorithm variants are Neuroevolution of Augmenting Topologies (NEAT) [114], which is used to evolve multilayer perceptrons and CoSyne [115] which is also used to evolve MLPs. These methods can be useful in creating controllers for online systems just like the CMAC, but must do so offline.
Figure B.1: Genetic Algorithm Flowchart - A flowchart describing the standard genetic algorithm process.
Appendix C

Calculating the Continuous CMAC Credit Assignment Lookup Tables

C.1 Creating the Lookup Tables

In order to calculate the credit assignment lookup tables for use in the continuous CMAC credit assignment method described in Chapter 7, a brute force method is employed. This method is described below.

C.1.1 Binary Combinations Representation

Each combination of the individual elements of $z$ can be represented as a bit vector. A bit set to ‘1’ represents that that particular $z_i$ value is used in that particular combination. In Table C.1 the bit vectors for $n_z = 5$ are shown.

The bit table can be generated computationally by noticing that there is a pattern when the $z$ combinations are written out in lexicographic ordering, as they are in Table C.1. One pattern is that in group one, there will be two bits set, in group two there will be three bits set, and so on.

Another pattern is that, initially for a group, for the first bit vector for the first row, all the bits set to ‘1’ will be as far left as possible. To get the second row the right-most bit that is set to ‘1’ is shifted one space to the right. For each successive row, that bit continues shifting right until it reaches the end. Then the next right-most bit that is set to ‘1’ is shifted one to the right, and the right-most bit is moved back to be next to the bit which has just moved. This pattern is repeated until all the set bits
### C. Calculating the Continuous CMAC Credit Assignment Lookup Tables

#### Table C.1: Vector Binary Representation of \( z \)

<table>
<thead>
<tr>
<th>idx</th>
<th>( z )</th>
<th>( z_1 )</th>
<th>( z_2 )</th>
<th>( z_3 )</th>
<th>( z_4 )</th>
<th>( z_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( z_1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( z_2 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( z_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>( z_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>( z_5 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Group 1**

<table>
<thead>
<tr>
<th>idx</th>
<th>( \min(\beta_1, \beta_2, \beta_3) )</th>
<th>( \min(\beta_1, \beta_3, \gamma_1) )</th>
<th>( \min(\beta_1, \beta_4, \gamma_2) )</th>
<th>( \min(\beta_1, \beta_5, \gamma_3) )</th>
<th>( \gamma_4 )</th>
<th>( \gamma_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 0 0 0 0</td>
<td>1 1 0 0 0 0 0</td>
<td>1 0 1 0 0 0 0</td>
<td>1 0 0 1 0 0 0</td>
<td>1 0 0 0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

**Group 2**

<table>
<thead>
<tr>
<th>idx</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \gamma_1, \gamma_2, \gamma_3) )</th>
<th>( \min(\beta_1, \beta_2, \beta_4, \gamma_1, \gamma_3, \gamma_4) )</th>
<th>( \min(\beta_1, \beta_2, \beta_5, \gamma_1, \gamma_4, \gamma_5) )</th>
<th>( \min(\beta_1, \beta_3, \gamma_2, \gamma_3, \gamma_4, \gamma_10) )</th>
<th>( \gamma_6 )</th>
<th>( \gamma_7 )</th>
<th>( \gamma_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 0 0 0 0</td>
<td>1 1 0 1 0 0 0</td>
<td>1 0 1 0 1 0 0</td>
<td>1 0 0 1 0 1 0</td>
<td>0 1 1 1 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Group 3**

<table>
<thead>
<tr>
<th>idx</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \beta_4, \gamma_1, \gamma_2, \gamma_3, \gamma_5, \gamma_6, \gamma_8) )</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \beta_4, \gamma_1, \gamma_2, \gamma_4, \gamma_5, \gamma_7, \gamma_9) )</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \beta_5, \gamma_1, \gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_10) )</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \beta_5, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9, \gamma_10) )</th>
<th>( \gamma_10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 0 0</td>
<td>1 1 1 0 1 0</td>
<td>1 1 0 1 1 1</td>
<td>0 1 1 1 1 1</td>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

**Group 4**

<table>
<thead>
<tr>
<th>idx</th>
<th>( \min(\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9, \gamma_10) )</th>
<th>( \gamma_10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 1</td>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

are pushed up against the right side, at which point the final bit vector for that group will have been generated.

196
C.1 Creating the Lookup Tables

C.1.2 Lookup Table for Combinations Table

As noted in Chapter 7, the combinations table from group two and higher can be populated entirely by calculating the minimums with three values (for each row) from a previous group. To do this, a lookup table is used to select (for a row being calculated) the three correct rows from the previous group to use in the minimum function.

Using the bit vectors in Table C.1, the indices of the three rows required can be calculated by masking out a set bit with zero in a bit vector three times, and finding the masked matching rows in the previous group. The matching rows are the indices that should be recorded.

For example, in Table C.1 consider finding the corresponding indices for row one in group two. The bit vector is given by $[1 \ 1 \ 0 \ 0]$. Masking out the first bit creates bit vector $[0 \ 1 \ 0 \ 0]$. This bit vector matches with row five in group one. Masking the second set bit gives $[1 \ 0 \ 0 \ 0]$ which matches row two in group one. Finally, masking the third set bit gives $[1 \ 0 \ 0 \ 0]$ which matches row one in group one. Thus, the minimum of the values at indices $\{4, 2, 1\}$ in group one can be used to calculate the equivalent value of $\min(z_1, z_2, z_3, \gamma_1, \gamma_2, \gamma_5)$ as in (C.1).

$$\min(z_1, z_2, z_3, \gamma_1, \gamma_2, \gamma_5) = \min(\min(z_1, z_5, \gamma_4), \min(z_1, z_3, \gamma_2), \min(z_1, z_2, \gamma_1))$$

(C.1)

This index calculating algorithm must be performed for all groups higher than group one in order to create the required lookup table.

It is only really feasible to calculate the lookup table up until a $z$ length of $n_z = 20$ or so, since the number of rows is given by $2^{n_z}$, and $n_z = 20$ is about the threshold at which the computational feasibility becomes too difficult, unless highly parallelized.

C.1.3 Lookup Table for Selection Table

The selection look up table can be calculated using the same bit vector representation of the combinations shown in Table C.1. Calculating this look up table is simple. If a bit is set at the interested $z_i$ selection, then that row index is recorded.

For example, for $z_2$ record all row indices in group one where the second bit is set as ‘1’. This gives $\{1, 5, 6, 7\}$. In group two the indices are $\{1, 2, 3, 7, 8, 9\}$. In group three the indices are $\{1, 2, 3, 5\}$. Finally, in group four the only index activated is index $\{1\}$.  

197
C. CALCULATING THE CONTINUOUS CMAC CREDIT ASSIGNMENT LOOKUP TABLES

To create the lookup table, perform this algorithm for all \( n_2 \) desired and for all \( z_i \), where again \( n_2 = 20 \) is probably the limit of computational feasibility.
References


REFERENCES


[77] Chia-Kuei Tsai. A hierarchical CMAC architecture for context dependent function approximation. In IEEE International Conference on Neural Networks, 1 pages 629–634 vol.1, jun 1996. 27.


REFERENCES


[106] C. Lauffer and G. Composer. Regularisation for the kernel recursive least squares CMAC. In Neural Networks (IJCNN), The 2018 International Joint Conference on, pages 1–8, 2012. 103, 118


