Development of a Simulation Environment for Signal Processing Algorithms

Sharat Ahuja

University of Rhode Island

Follow this and additional works at: http://digitalcommons.uri.edu/theses

Terms of Use
All rights reserved under copyright.

Recommended Citation
http://digitalcommons.uri.edu/theses/966

This Thesis is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Master’s Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons@etal.uri.edu.
DEVELOPMENT OF A SIMULATION ENVIRONMENT FOR SIGNAL PROCESSING ALGORITHMS

by

SHARAT AHUJA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

1988
MASTER OF SCIENCE THESIS
OF
SHARAT AHUJA

APPROVED:
Thesis Committee
Major Professor

Dean of Graduate School

UNIVERSITY OF RHODE ISLAND
1988
ABSTRACT

The development of a simulation environment for the study of signal processing algorithms is discussed. The project was divided into two parts. The first and the major part was the development of MATLAB, a matrix manipulation program, into a suitable simulation environment for studies in signal processing. The second part used the software tools developed by the first in performing a simulation study involving the use of state space models in parameter estimation.

Developing software tools in MATLAB included expanding the original inbuilt set of tools to enable the users to write their own functions effectively. This meant creating a set of sophisticated tools and adding these to the source of the MATLAB program. The method for adding functions to the MATLAB library for use by the whole programming community has been detailed. A large set of signal processing functions have been added to the MATLAB library, thus increasing the power of MATLAB as a signal processing simulation environment.

The problem of approximate realization which constitutes the second part of the project, has been dealt with at the simulation stage. Starting out with noisy data, the aim was to develop a state space model which approximated this data. Different ways of arriving at a solution have been considered. The results of the simulation studies are detailed.
ACKNOWLEDGEMENTS

I am grateful to my major professor, Dr. Richard J. Vaccaro for the valuable guidance he gave me and also for motivating and encouraging me throughout my work on my thesis.

I would like to thank my fellow graduate students who helped me at various times. In particular, I would like to thank Simon Hill and Alex Kot for their help whenever I needed it. My thanks to Dave Oliviera, now at Avanti Corporation, for helping me build up the MATLAB environment here at school.

Finally, I would like to thank my mother and sisters in India for the strength they have given me to complete my studies.
Contents

1 Introduction ................................................................. 1
   1.1 The problem in brief ........................................... 1
   1.2 Programming language classifications ....................... 2
   1.3 MATLAB - its superiority over others ...................... 3
   1.4 Overview of the thesis ........................................ 4
   1.5 Contributions of this thesis .................................. 5

2 Extended Functions .......................................................... 6
   2.1 User functions - a review .................................... 6
      2.1.1 Introduction ............................................ 6
      2.1.2 Usage .................................................. 6
   2.2 What are Extended functions? ................................. 9
      2.2.1 Usage .................................................. 9
   2.3 More with Extended functions ................................ 17

3 Signal Processing functions based on the FFT .................... 21
   3.1 FFT ................................................................... 22
   3.2 FFT related topics ............................................... 24
      3.2.1 Convolution .............................................. 24
      3.2.2 Correlation .............................................. 25
   3.3 Filters ............................................................. 25
      3.3.1 Lowpass Design ........................................... 26
      3.3.2 Bandpass Design ........................................ 28
# Building the Matlab library

## 4.1 The original setup

## 4.2 Starting out

## 4.3 Adding to the Help files

### 4.3.1 Summary

## 4.4 Adding Functions

### 4.4.1 Summary

## 4.5 Existing MATLAB functions

## 4.6 Current Signal Processing USER Functions

# Approximate Realization

## 5.1 Introduction

## 5.2 The problem of approximate realization

## 5.3 Simulation study

### 5.3.1 Simulation procedure

### 5.3.2 Simulation results

### 5.3.3 Simulation Efficiency

# Bibliography
List of Tables

2.1 Comparison table between "function" and native "MATLAB" calls. .................................................. 20

3.1 Numbers $\leq 10000$ as powers of 2 .................................................. 23

5.1 Method A: Correlated Noise Matrix .................................................. 54
5.2 Method B: Correlated Noise Matrix .................................................. 55
5.3 Method C: Correlated Noise Matrix .................................................. 55
5.4 Method A: Uncorrelated Noise Matrix .................................................. 55
5.5 Method B: Uncorrelated Noise Matrix .................................................. 55
5.6 Method C: Uncorrelated Noise Matrix .................................................. 55
5.7 Comparison table for simulation efficiency. .................................................. 56

I Numbers $\leq 10000$ with prime factors $\leq 23$. .................................................. 60
II Numbers $\leq 10000$ with prime factors $\leq 23$ (cont.). .................................................. 61
List of Figures

2.1 Sample "make" file ........................................... 17
4.1 MATLAB tree ................................................. 30
4.2 MATLAB character set .................................... 33
Chapter 1

Introduction

1.1 The problem in brief

Problem solving is a complex issue. To reach a solution, one can approach it in many ways. Our interest was in the solution of signal processing problems. Analysis is generally the first step in approaching a problem. However, another way to approach a problem is by a simulation study. Simulation studies can often tell us what needs to be analyzed. Simulation environments for the study of signal processing algorithms are available in plenty. However, most of the environments have their own limitations. Our intention was to develop an environment which would enhance the ease of doing simulation studies with signal processing algorithms that are described in the language of linear algebra. Such algorithms deal with arrays and matrices. Writing programs in FORTRAN or any other high level language involves accessing each element of the array or matrix, and this could be quite cumbersome. We were thus in search for a language which would enable us to eliminate this problem of accessing each memory cell and thereby make signal processing problem solving an easier task. MATLAB, a matrix manipulation program was found to have the desired qualities. A description of the MATLAB language is given later on in this chapter.
The project was divided into two parts. The first was to develop software tools which would automatically perform signal processing functions. The second was to use these tools in a simulation study of some signal processing algorithms. The software tools included building a set of signal processing functions which were to be incorporated as primitive MATLAB functions. The simulation studies would involve performance comparison between different state variable algorithms for parameter estimation and signal modeling.

1.2 Programming language classifications

Computer programming languages can be broadly classified into two kinds [1]. Statement oriented languages and languages oriented towards mathematical functions, the former called imperative languages and the latter applicative languages. FORTRAN, Cobol, Pascal, Ada are imperative languages, while APL, Speakeasy, MATLAB are applicative languages. Imperative languages get their name because of the dominant role played by imperative statements in such languages. The programs written in these languages have a ‘statement’ as a basic unit of work and the combination of such statements achieves the desired result in a program. Applicative or functional languages get their name from the prominent role played by mathematical functions and function applications. The capability of combining functions to produce more powerful functions is the strong feature of applicative languages.

Of the two, imperative languages are more efficient in terms of execution time because they reflect the structure and operations of the machine. However, they require the programmer to pay attention to machine level details. Programming using imperative languages is based on naming elementary memory cells, assignments to these cells and repetition of elementary operations. The functional programming style does not depend on these three actions, largely due to the simple and uniform data objects (like arrays, lists
etc.) which allow design of data structures without concern for elementary memory cells. Functional programming thus appears to be at a higher level than imperative programming.

1.3 MATLAB - its superiority over others

MATLAB is a powerful computer language for matrix manipulations and calculations [2,3]. Of the above mentioned programming language categories, MATLAB has the characteristics of applicative languages. It has only one data type; a rectangular matrix with complex elements. Scalars and vectors are special cases of this rectangular matrix. MATLAB is a program which can be worked in both the interactive and batch modes. Among the attractions of interactive working are immediate response and ease of correction of errors. The language is very easy to use and learn since a statement in MATLAB is almost identical to standard mathematical notation. A MATLAB program is easy to follow and one can trace out the path of the program in the first run by simply going through it. One of our objectives in choosing a language has been to consider the speed of writing a problem. The development time of writing a MATLAB program is small compared to writing it in any other high level language like FORTRAN. There are many reasons for this. A single data type in MATLAB eliminates complex data conversion statements and also eliminates equating elements of different data types as in other languages. As the language is based on the concept of arrays and matrices and processes these as entities, it eliminates the need for many of the loops necessary in other languages. The language also eliminates the need to dimension matrices. It is easy to append a matrix to another or to cut a larger matrix to a shorter one with any desired dimensions. The programs are thus easier to debug.

MATLAB has a large number of built in functions. An important feature of MATLAB is its built in capability for growth. It is easy to add functions written by the user to his or her own library or to the MATLAB
library for use by all users. MATLAB however is an interpreted language and thus is not very efficient. To solve a program with many loops takes a long time. However, as explained in chapter 4, this problem can be reduced, if not eliminated, with the help of Extended functions.

MATLAB is thus a valuable tool for most computer users even though it appears to be oriented towards the novice user. It is just like a “super desk calculator” although one with a vast memory and a rich and powerful vocabulary. It is intended to provide its users with the means to quickly formulate problems for computer processing and for obtaining answers in a minimum of time. MATLAB can thus be thought of as a “very high level” language.

1.4 Overview of the thesis

Chapter 2 deals with the introduction and formation of Extended functions. It contains a preview of the original User functions as provided by the original MATLAB package and also examples showing the difference between these two user defined functions.

Chapter 3 deals with signal processing functions based on the FFT. FFT related subroutines like convolution and correlation are described. It also deals with problems of filtering like lowpass filter, bandpass filter and filtering data using the transfer function and state variable approaches.

Chapter 4 deals with adding new functions and help files to the MATLAB library. The process of building the library is detailed. It also includes a list of the original MATLAB functions and a list of the new Signal Processing functions which have been added to the library.

Chapter 5 deals with an introduction to approximate realization and model reduction using the state space approach, and also the simulation example involving the use of state space models in parameter estimation.
1.5 Contributions of this thesis

- Gives a detailed discussion on writing and implementing extended functions for MATLAB. Extended functions were developed at URI to enhance the power of MATLAB.

- Documents 27 new signal processing functions which have been installed in the MATLAB language along with an on line HELP facility.

- Documents the procedures for installing new functions and HELP files into MATLAB.

- Demonstrates the usefulness of these tools through a simulation study of state space methods for parameter estimation. Simulation results are given which compare the performance of an existing method with two new methods.

- Documents the usage of the new signal processing functions added to the library through a “users manual” with help files and test cases. This document is available in the department.

- Documents the source code of the new signal processing functions added to the library through a “Matlab Signal Processing source manual”. The source is public domain and may be helpful to other users programming in FORTRAN. This manual is available with Dr. R.J. Vaccaro of the Electrical Engineering Dept.
Chapter 2

Extended Functions

2.1 User functions - a review.

2.1.1 Introduction

There are a few ways in which MATLAB can communicate with other programs. The MATLAB command "EXEC" helps a MATLAB program in communicating with other MATLAB programs. Communication with a FORTRAN program is done with the help of the "USER" function. There are two instructions that can be used to jump to a FORTRAN subroutine. One is the USER instruction and the other is the USRx instruction. The USER instruction was provided for in the original MATLAB package, whereas the USRx instruction was added later on to the same package to increase the number of such instructions. Both these functions are identical in all respects. The "x" in "USRx" can be any english letter from A to Z. Thus we can have FORTRAN subroutines with names USRA, USRB, .... USRZ.

2.1.2 Usage

In the MATLAB environment, the USER (or USRx) function can be used in the following way.
FUN1 = USER( FUN2, VAR1, VAR2)

where FUN2 is a matrix of dimensions MxN and VAR1 and VAR2 are scalars. FUN1 is the output matrix whose dimensions are specified in the USER function. Note that the USER function allows only one matrix and two scalars to be inputted. The major constraint in using this function is that FUN2 has to be a matrix of REAL numbers. The USER function does not work with complex numbers. On return, only the matrix FUN2 is read back into MATLAB and FUN1 is assigned the value of FUN2. The original value of FUN2 is not changed in the MATLAB environment.

The corresponding FORTRAN subroutine which needs to be written has to be in the following format:

```
SUBROUTINE USER(A,J,K,U,V)

where A represents the matrix FUN2 with dimensions JxK. U and V are scalars which represent VAR1 and VAR2. The input matrices and scalars have to be declared as DOUBLE PRECISION in the subroutine.

To illustrate the above, here is an example.

In the FORTRAN environment:

SUBROUTINE USER(A,O,P,E,F)
INTEGER*4 O,P
DOUBLE PRECISION A(O,P),E,F
A(1,1)=A(1,1)+E+F
O=1
P=1
RETURN
END
```

This subroutine has to be compiled first and then linked to MATLAB. A sample "make file" to enable compilation and linking of USER files is given
at the end of the next section in figure 2.2.1.

In the MATLAB environment:

\[ D = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \]

\[ S = 3 \]

\[ T = 7 \]

\[ EG = \text{USER}(D, S, T) \]

The value of EG would be 11.

We are adding the parameters E and F to \( A(1,1) \) in this simple example and then returning the value of \( A(1,1) \) back to the MATLAB environment. Note that initially the value of both \( O \) and \( P \) was 3, but in the subroutine, we change it to 1 at the end. This means that we are returning the matrix \( A \) with new dimensions 1x1 to the MATLAB environment. Also note that in the MATLAB environment, the dimensions of the matrix \( D \) is not specified in the call to the \text{USER} function. This is because the dimensions are picked up by the system inherently and there is no need for the user to specify them. This makes the Language very useful and powerful.

MATLAB allows the input matrix to the \text{USER} function to be redimensioned, so that the output matrix can be of a different dimension than the input matrix. This holds true with one constraint. The output matrix can only be smaller than the input matrix and not larger. If we have a matrix \( A \) of dimension 10x10 which is the input matrix to a \text{USER} function, the output matrix could have a dimension anywhere from 1x1 to 10x10, but not 10x11.

In order to remove the above mentioned constraints of the \text{USER} functions, the Extended functions were introduced into the MATLAB environment.
2.2 What are Extended functions?

To enhance the usability of the USER functions, the Extended functions were incorporated into the MATLAB environment. The Extended functions are depicted by "EXTx", where "x" is any english letter. The main differences between the USER and EXTx functions are:

- We can input as many matrices into the FORTRAN environment as we wish using the EXTx function, as against only one by the USER function.

- We can input as many scalar variables into the FORTRAN environment using the EXTx function, as against only two by the USER function. The scalar variables have to be dimensioned as 1x1 matrices in the EXTx function. So in effect, the scalar variables are special cases of matrices.

- A complex scalar, which is a special case of a complex matrix, can also be inputted. This would have to be declared as a COMPLEX*16 variable and will have to be dimensioned as 1x1. A real scalar can be declared as DOUBLE PRECISION.

- The input matrices and scalars can now contain complex numbers, as against only real numbers in the USER function.

- The output matrix from the EXTx function can be dimensioned smaller or larger than the input matrix, as against a smaller or equal dimensioned output matrix with the USER function. However, there is an upper limit to the dimensioning of the output matrix.

2.2.1 Usage

The EXTx function is defined in the following way in the MATLAB environment.

\[ \text{FUN1} = \text{EXTx(FUN2,FUN3,FUN4,A)} \]
where \( \text{FUN2, FUN3, FUN4} \) are input matrices and \( A \) is an input scalar. \( \text{FUN1} \) is the output matrix.

The corresponding \( \text{EXTx} \) statement in the \textsc{fortran} environment will be

\[
\text{SUBROUTINE EXTx(F1,M1,N1,F2,M2,N2,F3,M3,N3,E,M4,N4,RESULT,RROW,RCOL)}
\]

where \( F1, F2, F3 \) are the input matrices corresponding to \( \text{FUN2, FUN3, FUN4} \) with dimensions \( M1 \times N1, M2 \times N2, M3 \times N3 \) respectively and \( E \) is the input scalar variable corresponding to \( A \) with dimensions \( M4 \times N4 \). The output matrix returned to the \text{MATLAB} environment is returned in the matrix \( \text{RESULT} \) which has dimensions \( \text{RROW} \times \text{RCOL} \). Every subroutine using the extended function has to have the matrix \( \text{RESULT} \) declared within it. The values of \( \text{RROW} \) and \( \text{RCOL} \) have been specified in the \text{MATLAB} environment as 10000 and 1 respectively. The user need not specify the dimensions of \( \text{RROW} \) and \( \text{RCOL} \) at the beginning of the subroutine. However, the user needs to change the values of \( \text{RROW} \) and \( \text{RCOL} \) at the end of the subroutine to indicate to \text{MATLAB} the new dimensions of the \( \text{RESULT} \) matrix. An example would clarify all the points mentioned about extended functions.

All the input matrices have to be declared as \text{COMPLEX*16} even if they consist of real numbers. Input complex scalars (special case of a matrix) have to be declared as \text{COMPLEX*16} also. The input real scalars need to be declared as \text{DOUBLE PRECISION} and the output matrix \( \text{RESULT} \) has to be declared as \text{COMPLEX*16} in the subroutine of the extended function.

The extended function works in the following way. It can take as many input matrices as desired but returns only one. The desired output matrix is returned in the \( \text{RESULT} \) matrix. In the above description, \( \text{FUN1} \) is returned through \( \text{RESULT} \). The dimensions of \( \text{FUN1} \) can be smaller, equal to, or greater than that of \( \text{FUN2} \). The maximum number of elements in the input matrices or the returned output matrix can be 10000. This number was deemed sufficient to solve most problems in Signal Processing and hence
was chosen to be incorporated into the MATLAB extended function. The extended functions are typical “functions” and not “procedures” and thus return only one value.

As an example, let us say that we have an extended function EXTA which adds two matrices and returns the result. (We do not have to write a function to do this, as MATLAB does this quite simply. This is just an example to show how the extended function works.)

\[
\text{FUN2} = \begin{bmatrix} 2 \ + \ 6i \\ 5 \ - \ 3i \end{bmatrix}
\]

\[
\text{FUN3} = \begin{bmatrix} 2 \ + \ 3i \\ 4 \ + \ 2i \end{bmatrix}
\]

\[
\text{FUN1} = \text{EXTA}(\text{FUN2},\text{FUN3})
\]

The value of FUN1 on return would be

\[
\text{FUN1} = \begin{bmatrix} 4 \ + \ 9i \\ 9 \ - \ i \end{bmatrix}
\]

If we print FUN2, we would get

\[
\text{FUN2} = \begin{bmatrix} 2 \ + \ 6i \\ 5 \ - \ 3i \end{bmatrix}
\]

We see that the value of FUN2 is unchanged in the MATLAB environment.

Another example to illustrate the power of the extended function is as follows.

We have an input matrix A which is of dimension 2x2 and another matrix B which is also of dimension 2x2. We would like to place both these matrices side by side and multiply the resultant matrix with a scalar and then return this 2x4 matrix. This is how we would do it.

In the FORTRAN environment, we would have to write the following subroutine
SUBROUTINE EXTD(C,M1,N1,F,M2,N2,D,M3,N3,RESULT,RROW,RCOL)
INTEGER*4 M1,N1,M2,N2,M3,N3,RROW,RCOL,TOT,TOT1
COMPLEX*16 C(M1,N1),F(M2,N2),RESULT(RROW,RCOL)
DOUBLE PRECISION D
C Note that the matrix D has not been dimensioned and hence
C it could be considered as a scalar.
C
C Test to be sure that the result array will be smaller than
C the maximum allowed number of elements. The number of
C elements is passed in RROW.
C
TOT1 = M1*N1 + M2*N2
IF(TOT1 .GT. RROW)THEN
  WRITE(*,7)
  7 FORMAT(" ERROR FROM EXTD FUNCTION. MAXIMUM NUMBER " /
      & " OF OUTPUT ELEMENTS EXCEEDED.")
  RROW=1
  RETURN
END IF
C
C Copy array C into array RESULT
DO J= 1,M1
  DO I= 1,N1
    RESULT(J,I) = C(J,I)
  END DO
END DO
C Copy array F into array RESULT at the right place.
DO J= 1,M1
  DO I = N1+1,N1+N2
    RESULT(J,I) = F(J,I-N1)
  END DO
END DO
C Multiply the RESULT matrix with the scalar D.
DO I = 1,M1
   DO J = 1,N1+N2
      RESULT(I,J) = RESULT(I,J)*D
   END DO
END DO
TOT = N1+N2
C Indicate to MATLAB the returning size of the RESULT matrix.
C The returning size here is M1xTOT.
RROW=M1
RCOL=TOT
RETURN
END

The two lines
RROW=M1
RCOL=TOT
inform MATLAB that the row dimension of RESULT equals M1 and the column dimension RCOL equals TOT.

This subroutine has to be in a file extd.f and this file has to be compiled and then linked with MATLAB. A sample "make file" to achieve this is given at the end of this section in figure 2.2.1.

In the MATLAB environment, we have the following.

<> A = < 2 + 6i 5 + i
           5 - 3i 4 + 9i >
<> B = < 2 + 3i 3 + 4i
           -4 + 2i 8 - 2i >
<> C = EXTD(A,B,5)
The value of $C$ on return would be:

$$C = 
\begin{array}{cccc}
10 + 30i & 25 + 5i & 10 + 15i & 15 + 20i \\
25 - 15i & 20 + 45i & -20 + 10i & 40 - 10i
\end{array}$$

This example illustrates that the dimensions of the input matrix can be changed.

Another example which will clearly illustrate the usage of the extended function is the formation of a Hankel matrix from a data sequence. The FORTRAN subroutine is as follows.

```fortran
SUBROUTINE EXT_H(A,ROW,COL,M,M1,N1,N,M2,N2,RESULT,RROW,RCOL)
C Subroutine to formulate the hankel matrix from a data sequence.
C
C INPUT PARAMETERS:
C
C A - Complex matrix of dimensions ROWxCOL ( generally COL is equal to 1) of the input data sequence.
C M - Row dimensions of the output hankel matrix. The dimensions of M itself are M1xN1 (1x1).
C N - Column dimensions of the output hankel matrix. The dimensions of N itself are M2xN2 (1x1).
C
C OUTPUT PARAMETERS:
C
C RESULT - Complex matrix of dimensions MxN of the desired hankel matrix. RESULT is of initial dimensions RROW x RCOL.
C
INTEGER*4 ROW,COL,RROW,RCOL,M1,N1,N2
```
DOUBLE PRECISION M,N
COMPLEX*16 A(ROW,COL),RESULT(RROW,RCOL)
IP=IDNINT(M)
IQ=IDNINT(N)
CALL HANKELL(A,ROW,COL,IP,IQ,RESULT,RROW,RCOL)
RETURN
END

SUBROUTINE HANKELL(A,ROW,COL,IP,IQ,RESULT,RROW,RCOL)
INTEGER*4 ROW,COL,RROW,RCOL,IP,IQ
COMPLEX*16 A(ROW,COL),RESULT(IP,IQ)
C
C Test to be sure result array will be smaller than
C maximum allowed number of elements. The maximum number
C of elements is passed in RROW.
C
IF(IP*IQ .GT. RROW)THEN
    WRITE(*,50)
50 FORMAT("ERROR FROM HANK FUNCTION."
    & "MAXIMUM NUMBER OF ELEMENTS EXCEEDED.")
    RROW=1
    RETURN
END IF
C
C Generate Hankel matrix
C
ISUM=ROW+COL-1
DO 20 I = 1,IP
    DO 10 J = 1,IQ
        IS=I+J-1
        IF (IS .LE. ISUM) THEN
            \
RESULT(I,J)=A(IS,1)
ELSE
RESULT(I,J) = (0.,0.)
END IF
10 CONTINUE
20 CONTINUE

C Inform MATLAB of the new size of the matrix by returning
C the row and column sizes in the RESULT row and column
C parameters.
C
RROW=IP
RCOL=IQ
RETURN
END

In the Matlab environment, the program for using this extended function
should be as follows. Note that the input to this function could either be a
row or a column vector.

G = < 1 2 3 4 5 6 7 8 9 10 11 12 13 14 >
The call F = HANK(G,7,8) will result in

F =

1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11
5 6 7 8 9 10 11 12
6 7 8 9 10 11 12 13
7 8 9 10 11 12 13 14

We have created a 7x8 hankel matrix from the given data sequence.

A sample "make file" to compile and link files to MATLAB is given
here.
FFLAGS =
LDFLAGS =
FILES_OB = exta.o extd.o extz.o

files: $(FILES_OB)
matlink $(FILES_OB) -o matlab

Figure 2.1: Sample “make” file

2.3. More with Extended functions

MATLAB is an interpreted language and hence is quite inefficient if the program written consists of many DO Loops. But simulation studies require programs to collect statistical information from the intermediate and output data and this means calculating variances, means and other relevant statistical information. A MATLAB program containing just one DO Loop with the top limit as 100 could take as much as 5-6 minutes, depending on the number of computations it has to do inside the loop. A program with nested DO Loops could take a much longer time depending on the levels of nesting and computations to be done. This was a big waste of time and hence MATLAB was not being used as much as it should have been.

The Extended functions introduce a way to reduce this time consumption. A MATLAB program consisting of DO Loops could be written using an extended function to reduce the computation time. The DO Loops along with the computations inside the loop could be written in FORTRAN using an extended function. This function could then be called from the MATLAB program. Since the DO Loops now are compiled, they result in a tremendous improvement in the execution speed of the program. This result can be seen by running two programs, one with the DO Loops in MATLAB and the other with an extended function which achieves the same computations.
An example to illustrate this is given below.
Let us consider a FOR loop in a MATLAB program.

\[
\text{For } i=1:M, \text{ For } j=1:N, \ A(i,j) = 1/(i+j-1);
\]

This loop can be coded in FORTRAN and used with the extended function in the following way.

```fortran
SUBROUTINE EXTX(M,M1,N1,N,M2,N2,RESULT,RROW,RCOL)
INTEGER*4 M1,N1,M2,N2,RROW,RCOL,IM,IN
DOUBLE PRECISION M,N
COMPLEX*16 RESULT(RROW,RCOL)
C To correctly dimension the output matrix RESULT, we need
C to call another subroutine in the following way.
IM=IDINT(M)
IN=IDINT(N)
CALL SUBA(IM,M1,N1,IN,M2,N2,RESULT,RROW,RCOL)
RETURN
END
C
C SUBROUTINE SUBA(IM,M1,N1,IN,M2,N2,RESULT,RROW,RCOL)
INTEGER*4 M1,N1,M2,N2,RROW,RCOL,IM,IN
COMPLEX*16 RESULT(IM,IN)
DOUBLE PRECISION A
C
DO I=1,IM
  DO J=1,IN
    A=1.0/FLOAT(I+J-1)
    RESULT(I,J)=DCMPLX(A,0.0)
  END DO
END DO
END DO
```
C Indicate to Matlab the returning dimensions of the C RESULT matrix.

RROW=IM
RCOL=IN
RETURN
END

The call now in the MATLAB environment

<> B = EXTX(M,N)

would return the same result, though much quicker than the original MATLAB code. The values of M and N have to be specified before the call.

I have compared the cpu times of the above "function" call and the same calculation done in native "MATLAB" for different values of M and N. The results are produced in Table 2.3. Note that as the values of M and N increase, the "function" call time remains almost constant, whereas the "MATLAB" call time increases exponentially.

One goal of this thesis was to write a set of extended functions for many common signal processing tasks. These extended functions were given descriptive command names and incorporated into the MATLAB language itself. Thus they are available to any MATLAB user in either interactive or batch mode. In addition, on-line HELP files for these new functions were provided.

The documentation on extended functions given in this chapter gives all the information required for a user to write his or her own special purpose extended functions. We see that an extended function for a one line MATLAB code consists of quite a few lines of FORTRAN code. However, if the speed is desired, and your MATLAB program, which consists of this line is used often, it may be worth while to make use of the extended function.

The only disadvantage in using the Extended function for DO Loops could be while using matrices for computations inside the loops. The main purpose of using MATLAB, where you do not need to specify the dimensions of matrices during computations, is defeated. In the FORTRAN
I M N function MATLAB
5 5 0.4 1.2
10 10 0.3 3.8
15 15 0.2 8.9
20 20 0.3 16.5
25 25 0.2 28.7
50 50 0.4 3:13.4
100 100 0.8 34:17.7

Table 2.1: Comparision between “function” and native “MATLAB” calls.

subroutine, you would still have to access each data element of the matrix. Well, you can’t have the pie and eat it too!
Chapter 3

Signal Processing functions based on the FFT

Algorithms for computing the Fast Fourier Transforms (FFT) are available in plenty. They have been dealt with very thoroughly and a lot of methods for computing the DFT efficiently are available. We have, in our work, chosen an algorithm which justifies our choice for applications in Digital Signal Processing (DSP). The topics of convolution and correlation have been dealt with - the algorithms for these use the FFT algorithm we have chosen. A justification for using the FFT in such topics is detailed later on. This chapter also deals with problems of filtering. The lowpass and bandpass designs used here are quite standard [8] and we have explained how these designs were used to make up the MATLAB function library. The bandpass and lowpass filter designs use the filtering routines which filter data using the ARMA parameters. The function for filtering using the ARMA parameters has been included in the MATLAB library, but has not been detailed here. Its definition is given in chapter 4.
3.1 FFT

The use of Fast Fourier Transform algorithms in Digital Signal Processing became widespread as a result of the well known paper by Cooley and Tukey [16]. However, most of the applications were oriented towards using the value of N, the data sequence length, as a power of 2, as this decomposition lead to a highly efficient computation when using Cooley and Tukey's algorithm. For a majority of applications, this restricted choice of N was adequate. The need to have the value of N which was not a power of 2 was encountered by Singleton [13] in spectral analysis of speech and economic time series data. A mixed-radix FFT algorithm was developed by a lot of researchers but the one by Singleton [13] surpassed most others in terms of ease of use, flexibility and efficiency. We have thus considered the mixed-radix FFT algorithm by Singleton and have included it in the MATLAB library. Singleton FFT has been used very widely in a lot of applications due to its efficiency and flexibility in choosing different transform lengths [14].

Singleton's algorithm for computing the Fast Fourier Transform is based on the method proposed by Cooley and Tukey, but is a mixed radix algorithm. In his paper, Singleton presents an improved method of computing a transform step corresponding to an odd factor p of N. For details on how his algorithm works, the reader is referred to [13]. The definition of the transformation by Singleton is given as

$$\alpha_k = \sum_{j=0}^{n-1} x_j \exp(i2\pi jk/n)$$

for k=1,2,...n-1, where \{x_j\} and \{\alpha_k\} are both complex valued. The inverse transformation is given by

$$x_j = \sum_{k=0}^{n-1} \alpha_k \exp(-i2\pi jk/n)$$

for j=1,2,...n-1. Singleton has also provided an algorithm for computing the transform for real values, which uses the fact that the transform of real
Table 3.1: Numbers $\leq 10000$ as powers of 2

<table>
<thead>
<tr>
<th>2</th>
<th>32</th>
<th>512</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>64</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>4096</td>
<td></td>
</tr>
</tbody>
</table>

data has complex conjugate symmetry

$$\alpha_{n-k} = \alpha_k^*; \quad k = 1, 2, \ldots n - 1$$

Singleton's algorithm works for numbers $N$ whose prime factors are less than or equal to 23. I have produced two tables, one with values of $N$ with prime factors less than or equal to 23 and Table 3.1 with values of $N$ as powers of 2. The table with prime factors less than or equal to 23 can be found in the appendix. It has not been produced here as the number of entries in that table exceed 1300. Table 3.1 shows the wide jumps needed when a number falls in between two acceptable limits for FFTs which use algorithms with $N$ as powers of 2. Singleton's table, does not produce such large jumps and is thus more efficient to use. As an example, let us consider that an FFT of a data sequence of length 6024 has to be found out. From Table 3.1, we note that we would have to use a data length of 8192 for finding the FFT as 6024 is not a power of 2 and this algorithm uses only values of $N$ as powers of 2. In contrast, we see from Singleton's table that the next data length we need to jump to is 6048, which is a jump of 24 unlike a jump of 2168 when using Table 3.1. To reduce the size of the sequence would result in an inaccurate result as the jump in that direction would be very large too.

The maximum jump needed by Singleton's FFT algorithm is 55 in contrast to a huge maximum jump of 4095 by algorithms which use transform lengths with powers of 2. This itself justifies the use of Singletons's FFT.

The MATLAB function FFT has been written to perform the FFT. It is documented in chapter 4.
3.2 FFT related topics

3.2.1 Convolution

The convolution of two N point sequences $x_1(n)$ and $x_2(n)$ is denoted by

$$x_3(n) = \sum_{m=0}^{N-1} x_1(m)x_2(n - m)$$  \hspace{1cm} (3.1)

where $x_3(n)$ is the linear convolution of $x_1(n)$ and $x_2(n)$. $x_3(n)$ can have at most $2N - 1$ non zero points. The above convolution of 2 data sequences can be done directly using equation 3.1 or by using discrete fourier transforms to arrive at $x_3(n)$. Using discrete fourier transforms, $x_3(n)$ would be obtained by first finding the transforms of $x_1(n)$ and $x_2(n)$ for $2N - 1$ points and then finding the inverse transform of their product. The following equations describe this method.

$$X_1(k) = \sum_{n=0}^{2N-2} x_1(n)W_{2N-1}^{nk}$$

$$X_2(k) = \sum_{n=0}^{2N-2} x_2(n)W_{2N-1}^{nk}$$

$$x_3(n) = \frac{1}{2N - 1} \left[ \sum_{k=0}^{2N-2} [X_1(k)X_2(k)]W_{2N-1}^{-nk} \right] \Re_{2N-1}(n)$$

where $W_N = \exp(-j2\pi/N)$ is the twiddle factor and $\Re_N(n)$ is the rectangular sequence

$$\Re_N(n) = \begin{cases} 1 & 0 \leq n \leq N - 1 \\ 0 & \text{otherwise} \end{cases}$$

Due to the presence of highly efficient algorithms available for computing the discrete fourier transform of finite duration sequences, it has been found that using the transform method instead of the direct method would lead to savings in time as the number of multiplications would decrease by using the transform method. For short sequences, it might still be better to use the direct method of equation 3.1.

In building the convolution function for the MATLAB library, we have used only the transform method. The overhead by using the transform
method for short sequences as compared to the direct method is very small and hence the direct method has not been used. For extremely long sequences, the overlap add and the overlap save methods can be used [8,10,12]. However, we are dealing with a set of finite sequences. Real time data acquisition is not being considered, thus eliminating the use of very long sequences in our applications. MATLAB is an interactive program and the data has to be present to be acted upon. We have thus not used the overlap add or the overlap save methods in our programs.

The MATLAB function CONV has been written to perform convolution. It is documented in chapter 4.

### 3.2.2 Correlation

Calculating the autocorrelation estimate is similar to that of calculating the convolution of two sequences. The autocorrelation of the sequence $x(n)$ corresponds to the circular convolution of $x(n)$ with $x(-n)$. We can thus use the FFT to efficiently compute the autocorrelation estimate

$$C_{xx}(m) = \frac{1}{N} \sum_{n=0}^{N-|m|-1} x(n)x(n+m); \quad 0 \leq m \leq M-1 \quad (3.2)$$

where $M \leq N$ is the number of lags desired. The procedure using the Fast Fourier Transform is detailed in [8]. For small values of $M$, it may be more efficient to calculate equation 3.2 directly. However, we have only used the FFT to compute all correlation estimates. The justification for this is the same as that given in the convolution section.

The MATLAB function CORR has been written to perform correlation. It is documented in chapter 4.

### 3.3 Filters

In my approach to creating functions for Digital Filters, I have essentially used the same designs for the Butterworth lowpass filter and the band-
pass filter as described by Oppenheim and Schafer [8]. However, a few assumptions have been made and they are detailed ahead.

### 3.3.1 Lowpass Design

The squared magnitude function for an analog Butterworth filter is of the form

\[
|H_a(j\Omega)|^2 = \frac{1}{1 + (j\Omega/j\Omega_c)^{2N}}
\]  

(3.3)

where \( \Omega_c \) is the analog cutoff frequency of the filter. We know that as the value of \( N \), the order of the filter, increases, the characteristics of the filter become sharper. In the S plane, the equation 3.3 can be written as

\[
H_a(s)H_a(-s) = \frac{1}{1 + (s/j\Omega_c)^{2N}}.
\]

We see that the poles of the squared magnitude function are then given by

\[
s = (-1)^{1/2N}(j\Omega_c)
\]

which indicates that there are \( 2N \) poles equally spaced in angle on a circle of radius \( \Omega_c \) in the S plane. The poles are symmetrically located with respect to the imaginary axis, with no pole occurring on the imaginary axis and only poles appearing on the real axis for odd \( N \). The angular spacing of the poles on the circle is given by \( \pi/N \) radians. A factorization of \( H_a(s)H_a(-s) \) needs to be done to determine the transfer function of the analog Butterworth filter. As the poles of the magnitude squared function occur in pairs, we pick up only those poles which fall to the left of the imaginary axis in order to design a stable and causal filter.

Filtering the data has been done by designing 2\(^{nd}\) order sections from the poles of the analog butterworth filter and then using these sections in a cascaded manner. We see that if \( N \), the order of the filter is odd, then the filter has just one first order section given by

\[
\frac{\Omega_c}{s + \Omega_c}
\]

(3.4)
and \( m = (N - 1)/2 \) second order sections given by

\[
\frac{\Omega^2_c}{s^2 + 2\Omega_c \cos \left( \frac{\pi k}{N} \right) s + \Omega^2_c}; \quad k = 1, 2, \ldots m
\]  

(3.5)

If \( N \) is even, then we have \( m = N/2 \) second order sections given by

\[
\frac{\Omega^2_c}{s^2 + 2\Omega_c \cos \left( \frac{(2l-1)\pi}{N} \right) s + \Omega^2_c}; \quad l = 1, 2, \ldots m
\]  

(3.6)

The user provides a normalized digital frequency \( \alpha \) which is to be prewarped to \( \Omega_c \) for use in the equations 3.4, 3.5 and 3.6. The normalized digital frequency \( \alpha \) should have a value between 0 and 1 with both limits excluded. The prewarped cutoff frequency \( \Omega_c \) is given by

\[
\Omega_c = \frac{2}{T} \tan \left( \frac{\alpha \pi}{2} \right)
\]  

(3.7)

where \( T \) is the sampling period. The solution for \( \alpha \) from the above equation is

\[
\alpha = \frac{2}{\pi} \arctan \left( \frac{\Omega_c T}{2} \right)
\]

To map the above \( S \) plane equations into the \( z \) plane, we make use of the bilinear transformation which uses the equation

\[
s = \frac{2 \left( z - 1 \right)}{T \left( z + 1 \right)}
\]  

(3.8)

Equations 3.7 and 3.8 are then substituted in equations 3.4, 3.5 and 3.6 to arrive at a digital Butterworth lowpass filter design. Each second order section is then used to filter the input data. Filtering is done using a function which uses the ARMA parameters of each second order section generated by the above equations. Note that when equations 3.7 and 3.8 are substituted in equations 3.4, 3.5 and 3.6, \( T \), the sampling period disappears. We thus see that a change in the value of \( T \) has no effect on the design and thus we do not need to input the value of \( T \).

A HELP documentation of the subroutine LPF, which designs the lowpass filter as described above, is given in the chapter 4.
3.3.2 Bandpass Design

The traditional approach to the design of bandpass filters is to first design a frequency normalized prototype lowpass filter and then use an algebraic transformation, usually the spectral transformation, to derive the desired bandpass filter. We have used this approach in our design of the bandpass filter. Our methodology includes designing a lowpass butterworth filter as described in Section 3.3.1 with an normalized digital cutoff frequency of 0.25 and then using the spectral transformation to arrive at the appropriate design.

One thing to note while using the spectral transformations is that an \(N^{th}\) order lowpass filter produces a \(2N^{th}\) order bandpass filter. Thus if the user specifies an \(N^{th}\) order bandpass filter to be designed, a \(N/2^{th}\) order lowpass filter is designed and then a corresponding \(N^{th}\) order bandpass filter is constructed. In the event that \(N\) is odd, an \(\frac{N+1}{2}^{th}\) order lowpass filter is designed and correspondingly, an \((N + 1)^{th}\) order bandpass filter is constructed. This higher value of \(N\) has been chosen instead of a lower value as the filter characteristics of this higher order filter would be sharper than the desired filter, with no loss to the user.

The MATLAB function BAND has been written to design a bandpass filter. It is documented in chapter 4.
Chapter 4

Building the Matlab library

4.1 The original setup

The original MATLAB library by Cleve Moler [2] consists of 44 inbuilt functions. It is easy for a user to build up his own library of useful functions and link it with the original MATLAB library. As explained in chapter 2, this could be done with the help of the extended functions. Thus, a user could have a very powerful library of often used functions and thus eliminate the need to code in FORTRAN or MATLAB over and over again.

This chapter details building the MATLAB library in a Unix environment. However, the general methodology can be applied to building the library in any environment. The complete source code of the double precision version of MATLAB is made up by 3 libraries and a help unit. The MATLAB library tree is shown in figure 4.1. The unit MAT makes up the library 'libmatlab.a' and is located at the level /usr/matlab/matlab in the Unix environment we have on the Data General MV10000. The unit SYS makes up the library 'libsys.a' and is located at the level /usr/matlab/system. The unit LIB makes up the library 'libmatlib.a' and is located at the level /usr/matlab/library. The help unit is located at the level /usr/matlab/helper.

The subroutines in MAT make up the parser and the interpreter. The
subroutines in LIB include modified versions of the necessary routines from LINPACK, EISPACK and the BLAS. The subroutines in SYS are files dependent on the operating system. They involve file access and floating point word format. There are two new libraries included at this level. The USER and EXT libraries have been added locally to enhance the power of MATLAB so as to enable users to write their own functions and link it with MATLAB. The ‘libuser.a’ and ‘libext.a’ make up these libraries. The help unit consists of a help file which is accessed serially when needed. It also consists of a preprocessor which creates two files needed by the help utility.

4.2 Starting out

Suppose one wanted to add functions to the MATLAB library itself so that all users could have access to it. This, of course, would have to be a set of functions which would prove to be extremely helpful for the programming community on the whole. The functions would have to be tested very thoroughly to see that it handles all kinds of errors. The changes would then have to be done at the ‘root’ of the system where the original source code of MATLAB resides. This would have to be done by the system manager of the DG MV10000. It is suggested that before actually making changes at the root, the user should copy the complete source of the MATLAB program into his space and incorporate the functions there to verify if they
work the way they should. This would help make sure that the source code will not be altered in a wrong manner.

I have here detailed the way to add to the 'help' files of the utility and also to add functions to the original library. I have myself added quite a few DSP functions to the library. The 'make' facility of the Unix operating system has been utilized to make it easier to recreate the libraries to which the new functions have been added. Hence after modifying the desired files, compilations and links are made easier with the 'make' command. The appropriate commands are given in the respective sections.

4.3 Adding to the Help files

Adding to the help file is quite straightforward. Go into the top level 'matlab' directory and then go down one level to the 'helper' directory. Edit the file 'help'. One could either add in the middle or append to the file. The 'EOF' tag should remain at the end as it signifies the end of the help file. Just follow the same pattern as that of the current help file - the width of the help comments should be the same, as we do not want the lines to overlap on output. After the file is saved, run the Makefile in the helper directory by the following command.

    make index

This command creates 2 files - 'mathelp.idx' and 'mathelp.dac' at the /usr/lib level, which are used by the file 'helper.f' when help is needed. That is all that needs to be done to change the help file. We do not need to recompile or relink any other files. The original matlab executable file will still work in spite of the alteration to this 'help' file. The reason being that both 'mathelp.idx' and 'mathelp.dac' are serially accessed when the program is run. Another change which might need to be done is to change the size of a few arrays in the file 'helper.f'. This has been explained later on.
The help utility is set up in the following way. There is preprocessor called 'sethelp' which actually sets up this facility. It creates the two files 'mathelp.idx' and 'mathelp.dac' which are used in the utility by the subroutine 'helper.f'. The 'make index' call starts the preprocessor by executing it on the 'help' file and creating the 2 desired files at a different level. The file 'helper.f' contains the code for the output of the help facility. If one needs to change anything in this file, then it would have to be recompiled and relinked with the rest of the MATLAB files. It will generally not be necessary to change anything in this file except maybe the size of the arrays KEYS and LOCS. The present size of both these arrays is 128. The size of these arrays indicate the number of help calls in the help file. This will need to be changed if the number of calls go over 128. If the output of the help facility needs to be changed, then the code in this file will have to be altered. If 'helper.f' is changed, then one needs to go to the top level matlab directory and run the Makefile by the following simple command.

```make```

This would recompile 'helper.f' and relink it with the rest of the libraries in the MATLAB program.

4.3.1 Summary

To change the help file, do the following in the given order. The environment is the DG MV10000, under the Unix operating system.

```
cd /usr/matlab/helper
edit the 'help' file
make index
```

To change the helper.f file, do the following in the given order.

```
cd /usr/matlab/helper
edit file 'helper.f'
cd /usr/matlab
make
```
If a change is made to both the ‘help’ and the ‘helper.i’ files, then follow the respective procedures in the right order.

### 4.4 Adding Functions

A detailed description of adding functions to the MATLAB library is given here. The procedure detailed here requires editing quite a few files in the MATLAB source. Opening files and looking through it in the same order as I have detailed would clarify the procedure and lead to little or no confusion. A summary at the end of this section describes the procedure in brief.

MATLAB has a character set defined within it which has a one to one correspondence between a character or a special symbol and a number. The character set of MATLAB is given in figure 4.2.

Thus the number corresponding to the alphabet ‘A’ is 10 and ‘B’ is 11 and
The string 'ABS' is given by 10,11,28.

The names of functions in MATLAB can have a length of at most 4 letters. If one needs the function name to be of length smaller than 4 letters, then the rest of the letters must be replaced blanks. Thus the function name ABS corresponds to the string 10,11,28,36, as we see from the character set that 36 corresponds to a blank.

Adding new functions to the MATLAB library would include editing a few files and adding more files to the 'libmatlab.a' library. The way to add new functions is to first write extended functions which solve your problem. This is because the extended functions can now be incorporated into the MATLAB library by just a name change. If an extended function EXTC has to be changed to a function HIGH, the first thing to do is to change the subroutine call EXTC to HIGH. Then figure out the number sequence of HIGH from the character set given above. HIGH corresponds to 17,18,16,17. Now go into the second level matlab directory at level /matlab/m matlab. Edit file 'funs.f' and resize the arrays FUNN, FUNP and FUNL. Their sizes indicate the number of functions now present in the library. FUNN has been dimensioned as (4,112). This is because each function corresponds to 4 numbers, and at the time of writing, there were 112 functions in the library. By adding more functions, this number 112 would need to be changed in the variables FUNN, FUNP and FUNL. After this has been done, add the new functions to the comment list at the beginning of the file in the same way as the others are done. Four function names fit in one line and each line is numbered 1 through 9 with a $ following 9 and this repeats itself. Once this has been done, add the number sequence of the functions to the DATA statement of FUNN in the same order as that of the function names in the comment statements. Now go down to the DATA statement FUNP. The last number in this statement at the time of writing was 903. The number 03 indicates that three functions have been added to the MATLAB library. The number 9 indicates that the functions have been added by calling the subroutine MATFN9 which is present in
the file ‘matfn9.f’. In fact, all new functions which are to be added will be
called by this subroutine and so all numbers for new functions will start
with a 9. To add the new functions to the DATA statement of FUNP,
increment the number as 904,905 and so on, depending on the number of
functions you add. The first line of the DATA statement of FUNP should
be incremented by 3. Note that 7+3 is $ and also note that 9+3 is 2. We
discard the 1 at the beginning of 12. This is because we are dealing with
numbers from 1 to 9 and then $. We do not need to do anything else here.

After saving the file ‘funs.f’, go into the file ‘newfns.s’. This has been
written in DG assembly code and calls the desired new functions (subrou-
tines). The top of this file at the time of writing looks like the following.

.TITLE NEWFNS
.ENT NEWFNS
.EXTL HANK ATN FACT
.NREL 1

To add new functions to this list, add the name of the function to the end
of the '.EXTL' line. A new line with a '.EXTL' can be started after the
previous line is filled up. After this is done, go to the end of the file. It
looks like this.

; THE FOLLOWING TABLE CONTAINS THE ADDRESSES OF THE
; FUNCTIONS TO BE CALLED.

FADDR: .DWORD HANK
       .DWORD ATN
       .DWORD FACT

.END

Add the function name in the same way as the rest have been added, to
the end of the list. Save the file. The source code of the functions which are
being added to the library have to be included in the library ‘libmatlab.a’.
This will be done by 'make'ing the Makefile. The source code is to be put into a file and this file is to be added in the Makefile at this level. At the present time of writing, the three new functions HANK, ATN and FACT were put into the file 'funs9.f' and this file name is added to the Makefile. The Makefile need not be executed at this level. Go one level up to the top level 'matlab' directory and execute the Makefile there by the simple make command

```
make
```

This makes the main Makefile of the MATLAB program. It recreates the 'libmatlab.a' library and then creates a new executable MATLAB program.

### 4.4.1 Summary

To add new functions to the MATLAB library, do the following in the specified order.

```
cd /usr/matlab/matlab
edit file 'funs.f'
edit file 'newfns.s'
add new source file at this level
edit the Makefile at this level
cd /usr/matlab
make
```

### 4.5 Existing MATLAB functions

Included here is the original list of MATLAB functions which were provided in the original package. A list of functions which have been added to this original list are described in the next section.

- **INV(A)** - Inverse.
- **DET(A)** - Determinant.
• COND(A) - Condition number.
• RCOND(A) - A measure of nearness to singularity.
• EIG(A) - Eigenvalues and eigenvectors.
• SCHUR(A) - Schur triangular form.
• HESS(A) - Hessenberg or tridiagonal form.
• POLY(A) - Characteristic polynomial.
• SVD(A) - Singular value decomposition.
• PINV(A,eps) - Pseudoinverse with optional tolerance.
• RANK(A,eps) - Matrix rank with optional tolerance.
• LU(A) - Factors from Gaussian elimination.
• CHOL(A) - Factor from Cholesky factorization.
• QR(A) - Factors from Householder orthogonalization.
• RREF(A) - Reduced row echelon form.
• ORTH(A) - Orthogonal vectors spanning range of A.
• EXP(A) - e to the A.
• LOG(A) - Natural logarithm.
• SQRT(A) - Square root.
• SIN(A) - Trigonometric sine.
• COS(A) - Cosine.
• ATAN(A) - Arctangent.
• ROUND(A) - Round the elements to nearest integers.
• ABS(A) - Absolute value of the elements.
• **REAL(A)** - Real parts of the elements.

• **IMAG(A)** - Imaginary parts of the elements.

• **CONJG(A)** - Complex conjugate.

• **SUM(A)** - Sum of the elements.

• **PROD(A)** - Product of the elements.

• **DIAG(A)** - Extract or create diagonal matrices.

• **TRIL(A)** - Lower triangular part of A.

• **TRIU(A)** - Upper triangular part of A.

• **NORM(A,p)** - Norm with $p = 1, 2$ or "Infinity"

• **EYE(m,n)** - Portion of identity matrix.

• **RAND(m,n)** - Matrix with random elements.

• **ONES(m,n)** - Matrix of all ones.

• **MAGIC(n)** - Interesting test matrices.

• **HILBERT(n)** - Inverse Hilbert matrices.

• **ROOTS(C)** - Roots of polynomial with coefficients C.

• **DISPLAY(A,p)** - Print base p representation of A.

• **KRON(A,B)** - Kronecker tensor product of A and B.

• **PLOT(X,Y)** - Plot Y as a function of X.

• **RAT(A)** - Find "simple" rational approximation to A.

• **USER(A)** - Function defined by external program.
4.6 Current Signal Processing USER Functions

The following functions\(^1\) have been added by the process described above and are now a part of the MATLAB library. They can be called by any user and used in the same way as the original MATLAB functions.

- **HANK(S,M,N)** - Creates a Hankel matrix of order \((M,N)\) from a data sequence \(S\). The data sequence \(S\) can either be a column or a row vector.

- **IMSS(F,G,H,D,N)** - Finds the impulse response of a system using State Space variables. \(F,G,H\) and \(D\) are the State matrices and \(N\) is the length of the desired output data sequence (impulse response).

- **IMPZ(P,Z,G,N)** - Finds the impulse response of a system using Pole Zero variables. \(P\) and \(Z\) are the poles and zeroes of the system. The gain of the system is specified in \(G\). \(N\) is the length of the desired output data sequence (impulse response).

- **IMTF(B,A,N)** - Finds the impulse response of a system using Transfer Functions coefficients. The numerator coefficients are specified in "B" and the denominator coefficients are specified in "A". \(N\) is the length of the desired output data sequence (impulse response). The leading coefficients of \(B\) and \(A\) are inputted, even if they are 1.

- **NOSR(S,SNR)** - Adds real noise to the Data Sequence \(S\) with the desired SNR. The data sequence \(S\) can be either real or complex. The SNR is defined as

\[
SNR = 10 \log \frac{\sum_i ||a_i||^2}{\sigma^2 N}.
\]

\(^1\)The following functions are based on programs obtained from [9] by S.M. Kay: TSER, TSEC, LEVN, LERR and PSD.
• NOSC(S,SNR) - Adds complex noise to the Data Sequence S with the desired SNR. The data sequence S can be either real or complex. The definition of SNR is the given above.

• SOTA(A) - Sorts an array A of numbers in ascending order. The array A can be real or complex. Complex arrays are sorted in magnitude.

• SOTD(A) - Sorts an array A of numbers in descending order. The array A can be real or complex. Complex arrays are sorted in magnitude.

• ATN(A) - Finds the four quadrant arctangent of a vector or scalar A.

• TSER(A,B,N) - Generates a real AR, MA, or ARMA time series. A is an array of AR filter parameters. B is an array of MA filter parameters. The variance of real excitation noise is assumed to be 1.0.

• TSEC(A,B,N) - Generates a complex AR, MA, or ARMA time series. A is an array of AR filter parameters. B is an array of MA filter parameters. The variance of real and imaginary parts of the excitation noise is assumed to be 0.5.

• LEVN(R) - Implements the Levinsons recursion. R is an array of autocorrelation samples. On return, we get the solution of the Levinsons recursion.

• LERR(R) - Implements the Levinsons recursion. R is an array of autocorrelation samples. The prediction error powers are returned by this function.

• FFT(A,IS) - Finds the fast fourier transform of the data sequence in A. The value of IS indicates if the FFT or the inverse FFT has to be found. If IS is 1.0, the FFT is found and if the value of IS is -1.0, the inverse is found. The definition of the FFT is given in chapter 3.
• FACT(N) - Determines the next number closest to N which has a prime factor of 23 or less. This is to be used in conjunction with the FFT function as it uses the mixed radix transform for sequences whose length has prime factors less than or equal to 23.

• LPF(X,N,W) - Designs a lowpass Butterworth filter of order N and a normalized digital cutoff frequency W. The cutoff frequency has to be in the range 0 to 1.0, with both extremes excluded. X is the input real or complex data sequence. The definition of the filter is given in chapter 3.

• BAND(X,N,W1,W2) - Designs a bandpass filter of order N and normalized digital frequencies of W1 and W2. Both W1 and W2 have to be in the range 0 to 1, with both extremes excluded. X is the input real or complex data sequence which is to be filtered. The definition of this function is given in chapter 3.

• CONV(X,Y) - Convolve a sequence X with a sequence Y.

• CORR(X,M) - Find the autocorrelation of signal X. M is the required number of lags.

• PSD(A,B,N) - Compute a set of PSD values given the parameters of an ARMA model. Array A contains the AR parameters, array B contains the MA parameters. The variance of excitation noise is assumed to be 1.0. N determines the number of frequency samples desired.

• FITF(X,B,A,U) - Filter a data sequence X given the ARMA parameters. B holds the MA parameters and A holds the AR parameters. The leading coefficients of B and A are inputted even if they are 1.0. U contains the initial conditions.

• FISS(X,F,G,H,D,U) - Filter a data sequence X given the state matrices F, G, H and D. U contains the initial conditions.
• SSTF(F,G,H,D) - Converts the state space matrices F,G,H and D to the corresponding ARMA parameters. The first row of the returned matrix consists of the MA parameters and the second row, the AR parameters.

• WGNS(N,M,F) - Generates a real or complex white gaussian noise sequence in a matrix of dimension NxM. The generated real sequence has a variance of 1.0, while the complex sequence has a variance of 0.5 for both its real and imaginary parts. F is a flag which indicates if a real or complex noise is to be generated. If F is 0, real noise is generated, and if F is 1.0, complex noise is generated. The value of the 'seed' cannot be changed by the user. On entry, the value of the 'seed' is always 0 and then changes randomly.

• ACST(P,S,M) - Accumulates statistics when inserted in a loop. P is the input vector with any number of rows and one column consisting of the estimated parameters in the first column. S is the vector of the actual parameters. M is a matrix which has the accumulated statistics of these parameters. It should have the same number of rows as the matrix P, and two columns. The first column stores the accumulated sum of the estimated data in the right order and the second column stores the accumulated sum square of the difference between the estimated data and the actual input data. This function is used along with the function STAT which calculates the bias, variance and the mean square error of the estimated data. The call inside the loop should be of the form M = ACST(P,S,M). This function returns the accumulated data in the matrix M. The matrix M has to be initialized with zeroes outside the loop. This can be done with the help of the function ZERO.

• ZERO(M,N) - Creates an MxN matrix of all zeroes. An important use is for initializations of matrices.
* STAT(M,S,ITER) - Calculates the bias, variance and mean square error of a data set given the matrix M of accumulated data. The matrix M is generated with the help of the function ACST. S is a 1 column matrix of the actual data. ITER is the number of iterations over which the data has been accumulated. On return, this function returns a matrix of three columns, the first containing the bias of the estimated data, the second containing the variance of the estimated data and the third the mean square error.
Chapter 5

Approximate Realization

5.1 Introduction

The problems of model reduction and approximate realization for discrete-time, deterministic linear systems may be embedded into one another due to their similar nature. An introduction to solving these problems has been made using the state space models. It has been shown [11] that state space representation has certain advantages for approaching the problems of approximate realization and model reduction and we will be dealing with this approach here. Unlike the traditional scalar valued data dealt with by the autoregressive(AR), moving average(MA) or autoregressive moving average (ARMA) models, the state space approach deals with several variables simultaneously as vector valued variables. The AR, MA or ARMA models can deal with vector valued data, but the problems of model reduction and approximate realization are difficult to solve when compared with the state space approach. Theoretically, the Markovian or state space models are equivalent to the traditional ARMA type models because models in one representation can be transformed into another. However, there are some grounds in which these two models are not equivalent, like numerical stability, statistical properties of parameter estimates, ease of handling for vector valued or nonstationary time series represented in these two al-
ternative modes. We will not go into detail about the differences or the likes of these two modes here. The interested reader is referred to [11]. This reference deals with state space models for random processes. This chapter deals with the simpler problem of finding state space models for deterministic signals observed in noise.

5.2 The problem of approximate realization

The problem of approximate realization may be stated as follows. Given a data sequence, find a system whose impulse response approximates this data sequence in some sense. To explain this, let us construct a state space model of $p^{th}$ order. We will assume that the model is stable, i.e. the roots of the system are within the unit circle. The impulse response of the system can be described with the help of the state matrices $A$, $b$ and $c$ as

$$h_k = cA^{k-1}b$$

(5.1)

where $h_k$ is the impulse response for $k = 1, 2, \ldots$. The dynamic behaviour of the system is given by the following equations.

$$x_{k+1} = Ax_k + bu_k$$

$$y_k = cx_k$$

(5.2)

where $A$ is $p \times p$, $b$ is $p \times 1$ and $c$ is $1 \times p$.

The Hankel matrix of a system is formed from the impulse response as follows.

$$H_{m,n} = \begin{bmatrix}
h_1 & h_2 & \cdots & h_n \\
h_2 & h_3 & \cdots & h_{n+1} \\
\vdots & \vdots & \ddots & \vdots \\
h_m & h_{m+1} & \cdots & h_{m+n-1}
\end{bmatrix}$$

(5.3)
and using 5.1, can be written as

\[
\begin{bmatrix}
cb & cAb & \ldots & cA^{n-1}b \\
cAb & cA^2b & \ldots & cA^nb \\
\vdots & \vdots & \ddots & \vdots \\
cA^{m-1}b & \ldots & \ldots & cA^{m+n-2}b
\end{bmatrix}
\]

which reduces to

\[
\begin{bmatrix}
c \\
cA \\
cA^2 \\
\vdots \\
cA^{m-1}
\end{bmatrix}
\begin{bmatrix}
b & Ab & A^2b & \ldots & A^{n-1}b
\end{bmatrix}

\Theta_m

\]

(5.4)

where \( \Theta_m \) and \( C_n \) are the \( m \)-observability and \( n \)-controllability matrices for the system \((A, b, c)\). If the rank of \( \Theta \), the observability matrix is \( p \), where \( p \) is the dimension of the state vector of the model, then the model is said to be observable, or a pair of matrices \((A, c)\) is an observable pair. Similarly, a pair of matrices \((A, b)\) is controllable if the rank of the controllability matrix is equal to the order of the state vector. The Hankel matrix generated by the impulse response sequence \( h_k \) of a \( p^{th} \) order system has rank \( p \), for all \( m, n \geq p \). We can see from equation 5.4 that the factoring of the Hankel matrix could lead to specifying a state space model.

Given

\[
H_{m,n} = \Theta_{m,p} C_{p,n}
\]

(5.5)

where \( H_{m,n} \) is a Hankel matrix of rank \( p \); then a state space model which generates \( H \) is given by

\[
c = e_1^T \Theta = \text{first row of } \Theta
\]

(5.6)

\[
b = Ce_1 = \text{first column of } C
\]

(5.7)
where

\[ e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \]

To generate the state matrix \( A \) is not so trivial. For an infinite impulse response case,

\[ \Theta = \begin{bmatrix} c \\ cA \\ cA^2 \\ \vdots \end{bmatrix} \quad (5.8) \]

and

\[ \Theta^\dagger = \begin{bmatrix} cA \\ cA^2 \\ cA^3 \\ \vdots \end{bmatrix} \quad (5.9) \]

where \( \Theta^\dagger \) is a shifted up version of \( \Theta \). Note that

\[ \Theta A = \Theta^\dagger \]

and thus we see that

\[ A = \Theta^L \Theta^\dagger \]

where \( \Theta^L \) is the left inverse of \( \Theta \). For a finite impulse response case, given equation 5.5, we have

\[ \Theta_{m-1} A = \Theta_{m-1}^\dagger \quad (5.10) \]

where \( \Theta_{m-1} \) is the first \( m - 1 \) rows of \( \Theta \), and \( \Theta_{m-1}^\dagger \) is the \( m - 1 \) rows of \( \Theta \) starting from the second row. Note that both these matrices are of rank \( p \). We therefore have

\[ A = \Theta_{m-1}^L \Theta_{m-1}^\dagger \quad (5.11) \]

where \( \Theta_{m-1}^L \) is the left inverse of \( \Theta_{m-1} \). We see that

\[ \Theta_{m-1}^L \Theta_{m-1} = I_{p,p} \]
if $\Theta_{m-1}$ is of rank $p$. Thus the realization which corresponds to a factorization of $H$ is simply $(A, b, c)$ given by equations 5.6, 5.7 and 5.11. From equation 5.10, we see that both $\Theta_{m}$ and $\Theta^{\dagger}$ span the same $p$ dimensional vector space. We have here considered only single-input, single-output systems. For multivariate systems, the realization forms are similar.

A factorization which is of prime importance is the singular value decomposition (SVD) of $H$. On taking the SVD of $H$, its factorization is of the following form.

$$H = U\Sigma V^T$$

where $\Sigma$ is a matrix of singular values of $H$.

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \sigma_p \end{bmatrix}$$

with $\sigma_1 > \sigma_2 > \cdots > \sigma_p > 0$ and $U^TU = V^TV = I$. Any grouping of $U\Sigma V^T$ as $\Theta C$ would correspond to some $(A, b, c)$.

The problem of approximate realization can now be explained as described below. Given a data sequence $\{y_k\}$ for $k = 1, 2, \ldots N$, arrange $\{y_k\}$ in a Hankel matrix $H$ as

$$H_{m,n} = \begin{bmatrix} y_1 & y_2 & \cdots & y_n \\ y_2 & y_3 & \cdots & y_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ y_m & y_{m+1} & \cdots & y_{m+n-1} \end{bmatrix}$$

such that $m + n - 1 = N$. Assume that $m > n$. We now have a Hankel matrix $H_{m,n}$ of noisy data of rank $n$ with probability 1. We would like to get a rank $p$ Hankel matrix close to the noisy Hankel matrix of rank $n$.

An approximation desired as described is not easy to get. We can get a good rank $p$ approximation by taking the SVD of $H_{m,n}$ and keeping the first $p$ singular values and vectors. On taking the SVD, we have

$$H_{m,n} = U_{m,n} \Sigma_{n,n} V_{m,n}^T$$
Keeping the first \( p \) singular values and vectors, we get

\[
\hat{H}_{m,n} = \hat{U}_{m,p} \hat{\Sigma}_{p,p} \hat{V}_{p,n}^T
\]

This realization leads us to a rank \( p \) matrix \( \hat{H}_{m,n} \); Note that \( \hat{H}_{m,n} \) is not a Hankel matrix now. However, \( \|H - \hat{H}\| \) is minimal over all rank \( p \) matrices.

We could factorize \( \hat{H} \) as

\[
\hat{H} = \hat{U} \hat{\Sigma} \hat{V}^T
\]

and apply the realization formulae to get a correspondence to a state space model. Since \( \hat{H} \) is not a Hankel matrix,

\[
\hat{\Theta}_{m-1} A \neq \hat{\Theta}^\dagger_{m-1}
\]

for any \( A \). \( \hat{\Theta} \) and \( \hat{\Theta}^\dagger \) do not span the same \( p \) dimensional vector space. We can however say that

\[
\hat{\Theta}_{m-1} A = \hat{\Theta}^\dagger_{m-1} + E_1 \quad (5.13)
\]

where \( E_1 \) is an error matrix.

A least squares solution to this is proposed. If we can find \( E_1 \) to be minimum, then we can get a good approximation to \( A \). Thus, choosing \( A \) to minimize

\[
\|\hat{\Theta}_{m-1} A - \hat{\Theta}^\dagger_{m-1}\| \quad (5.14)
\]

is clearly a reasonable thing to do. This choice of \( A \) was first proposed by Kung [15] and has been used by many others [4,5,6,7].

However, another solution could be found by writing the equation 5.13 in the following way.

\[
(\hat{\Theta}_{m-1} + E_2) A = \hat{\Theta}^\dagger_{m-1} \quad (5.15)
\]

where \( E_2 \) denotes another error matrix. Thus, choosing \( B \) to minimize

\[
\|\hat{\Theta}_{m-1} - \hat{\Theta}^\dagger_{m-1} B\| \quad (5.16)
\]

would give us another good solution. Here \( A = B^{-1} \).
We note that the above solutions project columns of $\hat{\Theta}^\dagger$ onto the space spanned by $\hat{\Theta}$, or vice versa. This assumes that $\hat{\Theta}$ or $\hat{\Theta}^\dagger$ space is correct. But this is not correct as the noise affects both $\hat{\Theta}$ and $\hat{\Theta}^\dagger$. So we propose total least squares solution.

\[(\hat{\Theta}_{m-1} + E_1)A = \hat{\Theta}_{m-1}^\dagger + E_2\]  

(5.17)

such that

\[\|E_1;E_2\|\]

is minimized. The solution to this minimization problem is given in the next section.

Retracing our steps a bit, how would we know that the noisy Hankel matrix is to be approximated with a $p^{th}$ order Hankel matrix and not a $(p+1)^{th}$ order matrix. (Note that this $p^{th}$ order matrix is not really Hankel.) On taking the SVD of the noisy Hankel matrix $H$, we know that the $\Sigma$ matrix contains the singular values of the system. Thus

\[
\Sigma = \begin{bmatrix}
\sigma_1 & 0 & 0 & 0 \\
0 & \sigma_2 & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \sigma_n
\end{bmatrix}
\]

with $\sigma_1 > \sigma_2 > \cdots > \sigma_n > 0$, for $H_{m,n}$ with $m > n$. Also we note that $H_{m,n}$ is of rank $n$.

In theory, the rank of a Hankel matrix is equal to the number of its non zero singular values. If we have a noisy Hankel matrix, all its singular values will be positive. In order to approximate this Hankel matrix to a smaller rank matrix, we have to determine if $\sigma_i \gg \sigma_{i+1}$ for all $i = 1, 2, \ldots n$. Methods for determining this are well established [11]. Thus if $\sigma_p \gg \sigma_{p+1}$, then the rank $p$ matrix $\hat{H}_{m,n}$ will nearly be Hankel. Thus we get a nearly Hankel matrix of rank $p$ which is a good approximation to $H$, and so the above procedure will yield a $p^{th}$ order model which is nearly optimal with respect to the spectral norm of the Hankel matrix $H$. In addition, we thus
see that a disparity in the magnitude of the singular values \((\sigma_p \gg \sigma_{p+1})\) gives an indication of the model order to which the given system can be reduced without much change in the input-output behaviour.

5.3 Simulation study

The problem of approximate realization has been dealt with in the simulation study. The theoretical concepts have been detailed in the previous section. Given a data sequence, the aim is to find a system whose impulse response approximates this data sequence in some sense. The parameters of interest are the angles of the poles of the system. The given data sequence is noisy. The methods given in the last section for solving the \(A\) matrix of a state space model were used. The eigenvalues of \(A\) are the poles of the system.

**METHOD A**

This method uses equation 5.14 to get the desired approximate model. The solution to this equation can be written as

\[
A = \hat{\Theta}^L_{m-1} \hat{\Theta}^\dagger_{m-1}
\]  

(5.18)

\(\hat{\Theta}_{m-1}\) and \(\hat{\Theta}^\dagger_{m-1}\) are formed as mentioned in the previous section. \(\hat{\Theta}^L_{m-1}\) is the left inverse of \(\hat{\Theta}_{m-1}\). The MATLAB notation for solving the left inverse is a simple backslash. Thus, in MATLAB, equation 5.18 can be solved as

\[
A = \hat{\Theta}_{m-1} \backslash \hat{\Theta}^\dagger_{m-1}
\]

**METHOD B**

Equation 5.16 is used to get the desired approximate model. This equation can be written as

\[
B = \hat{\Theta}^\dagger_{m-1} \hat{\Theta}_{m-1}
\]  

(5.19)
\( \Theta_{m-1}^\dagger \) is the left inverse of \( \Theta_{m-1}^\dagger \). The MATLAB notation for equation 5.19 is

\[
B = \Theta_{m-1}^\dagger \backslash \Theta_{m-1}
\]

and the desired model is given by

\[
A = B^{-1}.
\]

**METHOD C**

This considers the least squares solution described in the previous section. Equation 5.17 is used to get the desired solution. We can rewrite the equation 5.17 as

\[
(\hat{\Theta}_{m-1} + E_1) A - (\hat{\Theta}_{m-1}^\dagger + E_2) = 0
\]

which equals

\[
[\hat{\Theta}_{m-1}^\dagger + E_1 : \hat{\Theta}_{m-1}^\dagger + E_2] [\begin{array}{c} A \\ -I \end{array}] = 0.
\]

The partitioned coefficient matrix in the above equation has dimensions \((m - 1) \times 2p\). In order for this matrix to have a \(p\) dimensional null space as shown in the above equation, it must have rank \(p\). Thus the matrices \(E_1\) and \(E_2\) represent perturbations which make the rank of the coefficient matrix equal to \(p\). The resulting rank \(p\) matrix corresponding to \([E_1:E_2]\) of minimum Frobenius norm is obtained from the following SVD

\[
[\hat{\Theta}_{m-1}^\dagger : \hat{\Theta}_{m-1}^\dagger] = U_{2p} \Sigma_{2p} V_{2p}^T.
\]

The smallest \(p\) singular values are set to zero, and only the first \(p\) columns and rows of \(U_{2p}\) and \(V_{2p}^T\) are retained resulting in the optimal rank \(p\) approximating matrix \(U \Sigma V^T\). The solution then proceeds as follows.

\[
U \Sigma V^T \left[\begin{array}{c} A \\ -I \end{array}\right] = 0
\]
This would give us the following equation to solve.

\[ V^T \begin{bmatrix} A \\ -I \end{bmatrix} = 0 \]

\[ [V_{11}^T V_{12}^T] \begin{bmatrix} A \\ -I \end{bmatrix} = 0 \]

and we could get the desired model as

\[ V_{11}^T A = V_{12}^T \]

In the MATLAB notation, equation 5.20 could be written as

\[ A = (V_{11}^T)^{-1} V_{12}^T \]

5.3.1 Simulation procedure

The procedure to find an approximate model from the given exact model is as follows. The exact model is given by a set of poles and zeroes which characterizes the desired system.

- Generate the impulse response of the system from the poles and zeroes using the function IMTF.
- Add noise with a desired SNR to the data sequence inside the loop using function NOSC, which adds complex noise to the desired data sequence.
- Form a Hankel matrix from the noisy sequence inside the loop using the function HANK.
- Use methods A,B and C to find out the poles from the generated approximate model.
- Compare the bias, variance and mean square error of the angles of the generated poles.
5.3.2 Simulation results

Noise has been added to the data sequence in two ways. Adding noise to the data sequence and then forming the Hankel matrix has been one of the ways. We call this the ‘correlated noise matrix’ method since the additive noise matrix has Hankel structure. The other way has been to form a matrix of uncorrelated noise and adding this to the Hankel matrix of the data sequence to form the noisy matrix, which is now no longer Hankel. We call this the ‘uncorrelated noise matrix’ method. This method generates data similar to that in an array processing problem where the noise present at each sensor is uncorrelated with the noise at other sensors, and the noise at a single sensor is uncorrelated at different instants of time.

The system used for this simulation had poles at 0.6108 and 0.7854 radians, both with radius 0.99. The length of each data sequence generated was 20 points. The number of trials in this study was 100.

‘Bias’ is the absolute difference between the sample mean and the actual value. ‘Variance’ is defined as the sum square of the difference between the estimated samples and the actual data over the number of iterations. ‘Mean Square Error’ (MSE) is the sum of the bias square and the variance.

The tables 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6 summarize the results obtained from methods A, B and C. The SNR is the Signal to Noise ratio in decibels. Var is the variance of the observed data and MSE is the Mean Square Error. We have only accumulated statistics of the angles $\Theta_1$ and $\Theta_2$ of the poles of the system.

From the study, the preliminary conclusion is that none of the methods
### Table 5.2: Method B: Correlated Noise Matrix

<table>
<thead>
<tr>
<th>SNR</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>$8 \times 10^{-4}$</td>
<td>$13.0 \times 10^{-3}$</td>
<td>0.105</td>
<td>0.232</td>
<td>$2.3 \times 10^{-3}$</td>
<td>$2.5 \times 10^{-3}$</td>
<td>0.019</td>
<td>0.489</td>
</tr>
<tr>
<td>Var</td>
<td>$0.231 \times 10^{-4}$</td>
<td>$2.2 \times 10^{-3}$</td>
<td>0.201</td>
<td>0.493</td>
<td>$0.173 \times 10^{-3}$</td>
<td>$1.6 \times 10^{-3}$</td>
<td>0.116</td>
<td>0.665</td>
</tr>
<tr>
<td>MSE</td>
<td>$2.378 \times 10^{-5}$</td>
<td>$2.369 \times 10^{-3}$</td>
<td>0.212</td>
<td>0.547</td>
<td>$1.781 \times 10^{-4}$</td>
<td>$1.606 \times 10^{-3}$</td>
<td>0.117</td>
<td>0.904</td>
</tr>
</tbody>
</table>

### Table 5.3: Method C: Correlated Noise Matrix

<table>
<thead>
<tr>
<th>SNR</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>0</td>
<td>$5.2 \times 10^{-3}$</td>
<td>0.010</td>
<td>0.235</td>
<td>3.1</td>
<td>$10^{-3}$</td>
<td>$4.1 \times 10^{-3}$</td>
<td>0.019</td>
</tr>
<tr>
<td>Var</td>
<td>$0.220 \times 10^{-3}$</td>
<td>$2.5 \times 10^{-3}$</td>
<td>0.202</td>
<td>0.491</td>
<td>$0.177 \times 10^{-3}$</td>
<td>$1.8 \times 10^{-3}$</td>
<td>0.116</td>
<td>0.669</td>
</tr>
<tr>
<td>MSE</td>
<td>$0.220 \times 10^{-3}$</td>
<td>$7.700 \times 10^{-3}$</td>
<td>0.212</td>
<td>0.546</td>
<td>$1.868 \times 10^{-4}$</td>
<td>$1.817 \times 10^{-3}$</td>
<td>0.117</td>
<td>0.906</td>
</tr>
</tbody>
</table>

### Table 5.4: Method A: Uncorrelated Noise Matrix

<table>
<thead>
<tr>
<th>SNR</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>$2.1 \times 10^{-3}$</td>
<td>0.014</td>
<td>0.003</td>
<td>0.176</td>
<td>2.9</td>
<td>$10^{-3}$</td>
<td>0.013</td>
<td>$7.4 \times 10^{-3}$</td>
</tr>
<tr>
<td>Var</td>
<td>$0.102 \times 10^{-3}$</td>
<td>0.001</td>
<td>$7.3 \times 10^{-3}$</td>
<td>0.284</td>
<td>$0.104 \times 10^{-3}$</td>
<td>0.001</td>
<td>$7.0 \times 10^{-3}$</td>
<td>0.435</td>
</tr>
<tr>
<td>MSE</td>
<td>$1.101 \times 10^{-4}$</td>
<td>0.001</td>
<td>$7.311 \times 10^{-3}$</td>
<td>0.315</td>
<td>$1.125 \times 10^{-4}$</td>
<td>0.001</td>
<td>$7.055 \times 10^{-3}$</td>
<td>0.469</td>
</tr>
</tbody>
</table>

### Table 5.5: Method B: Uncorrelated Noise Matrix

<table>
<thead>
<tr>
<th>SNR</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>$1.0 \times 10^{-3}$</td>
<td>$4.000 \times 10^{-4}$</td>
<td>0.018</td>
<td>0.185</td>
<td>2.0</td>
<td>$10^{-4}$</td>
<td>0.001</td>
<td>$8.1 \times 10^{-3}$</td>
</tr>
<tr>
<td>Var</td>
<td>$0.112 \times 10^{-3}$</td>
<td>$1.400 \times 10^{-3}$</td>
<td>0.007</td>
<td>0.281</td>
<td>$0.107 \times 10^{-3}$</td>
<td>0.001</td>
<td>$7.0 \times 10^{-3}$</td>
<td>0.489</td>
</tr>
<tr>
<td>MSE</td>
<td>$1.131 \times 10^{-4}$</td>
<td>$1.400 \times 10^{-3}$</td>
<td>0.007</td>
<td>0.315</td>
<td>$1.070 \times 10^{-4}$</td>
<td>0.001</td>
<td>$7.066 \times 10^{-3}$</td>
<td>0.533</td>
</tr>
</tbody>
</table>

### Table 5.6: Method C: Uncorrelated Noise Matrix

55
has an edge over the others in solving this problem. The reason that the total least squares solution does not give a better result than the other two is because the noise in the matrices $\hat{\Theta}_{m-1}$ and $\hat{\Theta}_m^\dagger$ is correlated. More analysis needs to be done to arrive at the right solution. We have shown that simulations can be done using the tools available to us in the MATLAB environment.

### 5.3.3 Simulation Efficiency

The above simulation study has been done using some of the new Digital Signal Processing functions which have been added to the MATLAB library. A comparison between writing pure MATLAB code without any of the new functions for doing the above simulation study and code with the new functions is made here. The saving in time is tremendous. Table 5.3.3 shows the execution time taken by these two methods for 100 iterations. Method 1 is the program with the new functions and Method 2 is the pure MATLAB code. The $cpu$ time is approximately equal to the sum of the User and System time. The $cpu$ time for the pure MATLAB program without any functions is equal to almost 27 minutes, whereas for the other program with functions it is 4 minutes. This comparison clearly shows the difference between the previous MATLAB language and the present one.

<table>
<thead>
<tr>
<th></th>
<th>Method 1</th>
<th>Method 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>16:49.9</td>
<td>1:22:12.0</td>
</tr>
<tr>
<td>User</td>
<td>3:52.0</td>
<td>25:59.5</td>
</tr>
<tr>
<td>Sys</td>
<td>6.4</td>
<td>50.9</td>
</tr>
</tbody>
</table>

Table 5.7: Comparison table for simulation efficiency.
RECOMMENDATIONS

MATLAB was originally designed to be a general purpose matrix manipulation program, for interactive use, to enable the user to arrive at a quick solution to his problem. As has been shown, it can now be used as a strong tool for solving Signal Processing functions, both in the interactive and batch modes. This has been possible due to the introduction of 'extended' functions into the MATLAB library. The language can still be further enhanced to make it a more powerful utility than what it is now. The present 'extended' functions return only one matrix and this is a drawback quite often. We need to have 'super extended' functions which would return more than one matrix. It may be possible to construct the 'super extended' functions in a similar way to the extended functions. It is thus recommended that the 'super extended' functions should be created.

It was noticed near the end of the project that calls to the newly added functions would 'bomb out' when the number of arguments in the function call were less or more than the desired number of arguments needed. A routine which returns error messages when such function calls are encountered is recommended to be added to the source of the MATLAB language. This routine may have to be written in DG assembly code.

MATLAB is an interpreted language, unlike many other languages. This effects the efficiency of the program. It is possible to construct a compiler for the MATLAB language. This would boost its usefulness tremendously. It would then be definitely a 'super high level' language. Building a compiler is another recommendation of this thesis.
References


<table>
<thead>
<tr>
<th>Numbers</th>
<th>prime factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 3</td>
</tr>
<tr>
<td>4</td>
<td>1, 2</td>
</tr>
<tr>
<td>5</td>
<td>1, 5</td>
</tr>
<tr>
<td>6</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>7</td>
<td>1, 7</td>
</tr>
<tr>
<td>8</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>9</td>
<td>1, 3</td>
</tr>
<tr>
<td>10</td>
<td>1, 2, 5</td>
</tr>
<tr>
<td>11</td>
<td>1, 11</td>
</tr>
<tr>
<td>12</td>
<td>1, 2, 3, 6</td>
</tr>
<tr>
<td>13</td>
<td>1, 13</td>
</tr>
<tr>
<td>14</td>
<td>1, 2, 7</td>
</tr>
<tr>
<td>15</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>16</td>
<td>1, 2, 4, 8</td>
</tr>
<tr>
<td>17</td>
<td>1, 17</td>
</tr>
<tr>
<td>18</td>
<td>1, 2, 3, 9</td>
</tr>
<tr>
<td>19</td>
<td>1, 19</td>
</tr>
<tr>
<td>20</td>
<td>1, 2, 5, 10</td>
</tr>
<tr>
<td>21</td>
<td>1, 3, 7</td>
</tr>
<tr>
<td>22</td>
<td>1, 2, 11</td>
</tr>
<tr>
<td>23</td>
<td>1, 23</td>
</tr>
<tr>
<td>24</td>
<td>1, 2, 3, 4, 8</td>
</tr>
<tr>
<td>25</td>
<td>1, 5</td>
</tr>
<tr>
<td>26</td>
<td>1, 2, 13</td>
</tr>
<tr>
<td>27</td>
<td>1, 3, 9</td>
</tr>
<tr>
<td>28</td>
<td>1, 2, 4, 7</td>
</tr>
<tr>
<td>29</td>
<td>1, 29</td>
</tr>
<tr>
<td>30</td>
<td>1, 2, 3, 5, 10</td>
</tr>
<tr>
<td>31</td>
<td>1, 31</td>
</tr>
<tr>
<td>32</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>33</td>
<td>1, 3, 11</td>
</tr>
<tr>
<td>34</td>
<td>1, 2, 17</td>
</tr>
<tr>
<td>35</td>
<td>1, 5, 7</td>
</tr>
<tr>
<td>36</td>
<td>1, 2, 3, 4, 9</td>
</tr>
<tr>
<td>37</td>
<td>1, 37</td>
</tr>
<tr>
<td>38</td>
<td>1, 2, 19</td>
</tr>
<tr>
<td>39</td>
<td>1, 3, 13</td>
</tr>
<tr>
<td>40</td>
<td>1, 2, 4, 5, 10</td>
</tr>
<tr>
<td>41</td>
<td>1, 41</td>
</tr>
<tr>
<td>42</td>
<td>1, 2, 3, 7, 14</td>
</tr>
<tr>
<td>43</td>
<td>1, 43</td>
</tr>
<tr>
<td>44</td>
<td>1, 2, 4, 11</td>
</tr>
<tr>
<td>45</td>
<td>1, 3, 5, 15</td>
</tr>
<tr>
<td>46</td>
<td>1, 2, 23</td>
</tr>
<tr>
<td>47</td>
<td>1, 47</td>
</tr>
<tr>
<td>48</td>
<td>1, 2, 3, 4, 8</td>
</tr>
<tr>
<td>49</td>
<td>1, 7</td>
</tr>
<tr>
<td>50</td>
<td>1, 2, 5, 10</td>
</tr>
<tr>
<td>51</td>
<td>1, 51</td>
</tr>
<tr>
<td>52</td>
<td>1, 2, 26</td>
</tr>
<tr>
<td>53</td>
<td>1, 53</td>
</tr>
<tr>
<td>54</td>
<td>1, 2, 3, 9, 18</td>
</tr>
</tbody>
</table>

Table I: Numbers less than 10000 with prime factors less than 23

60
| Table II: Numbers ≤ 10000 with prime factors ≤ 23 (cont.) |
Appendix A

Bibliography


