Evaluation of Compilers for MATLAB- to C-Code Translation

Master’s Thesis in Computer Systems Engineering
Markus Müllegger

School of Information Science, Computer and Electrical Engineering
Halmstad University
Evaluation of Compilers for MATLAB- to C-Code Translation

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

January 2008
Acknowledgement

You were born an original. Don’t die a copy.
∼John Mason

Thank you, Helmut, Maria and Tanja Müllegger, my family; for all your love and support during all my life, to finally let me be, where I am now.

I would like to thank Peter Brauer for letting me sit at Ericsson AB; in that way I could gain on practical experience for my thesis and personally. Furthermore thanks to Peter, for his forthcoming help and support during all my thesis work. Ulf Lindgren, special thanks to him, for this big amount of time he has spent, to explain things to me in detail and to proofread my thesis. The quality of my thesis gained considerably through Ulf. Big thanks to the rest of the baseband research team at Ericsson AB, for helping me out with all my questions.

Jerker Bengtsson, my junior supervisor, thanks a lot to him for spending so much time to answer my questions and to proofread my thesis. A thank you to Bertil Svensson, for his experienced supervision and being here when it was most needed. I would also like to thank Lene Nordrum, to proofread my thesis with her English profession.

Big thanks to Stefan Wegenkittl, my supervisor at Salzburg University of Applied Sciences, for his great advise and feedback on my work.

A thanks to Catalytic Inc. and Marc Barberis for the support and cooperation with MCS. Thank you to Fredrik Rodin and MathWorks for the help and collaboration with EMLC.
Abstract

MATLAB to C code translation is of increasing interest for science and industry. In detail two MATLAB to C compilers denoted as Matlab to C Synthesis (MCS) and Embedded MATLAB C (EMLC) have been studied. Three aspects of automatic code generation have been studied; 1) generation of reference code; 2) target code generation; 3) floating-to-fixed-point conversion. The benchmark code used aimed to cover simple up to more complex code by being viewed from a theoretical as well as practical perspective. A fixed-point filter implementation is demonstrated. EMLC and MCS offer several fixed-point design tools. MCS provides a better support for C algorithm reference generation, by covering a larger set of the MATLAB language as such. More suitable for direct target implementation is code generated from EMLC. As a result of the need to guarantee that the EMLC generated C-code allocates memory only statically, MATLAB becomes more constraint by EMLC. Functional correctness was generally achieved for each automatic translation.
Contents

Acknowledgement ii

Abstract iii

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Compilers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.1.1 The compiler process . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.1.1.1 Compiler front-end . . . . . . . . . . . . . . . . . . . . 6

1.1.1.2 Compiler back-end . . . . . . . . . . . . . . . . . . . . 7

1.1.2 MATLAB-C in contrast to C-Assembly . . . . . . . . . . . . . . 7

1.2 Problem definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

1.2.1 Compiler versus Interpreter . . . . . . . . . . . . . . . . . . . . 9

1.2.2 MATLAB to C synthesis issues . . . . . . . . . . . . . . . . . . . 11

1.2.3 Scope of investigation . . . . . . . . . . . . . . . . . . . . . . . . . 14

1.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

1.3.1 MCC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

1.3.2 The FALCON project from De Rose . . . . . . . . . . . . . . . . 18

1.3.3 MaJIC an alternative to the MATLAB interpreter . . . . . . . . 21

1.3.4 RTExpress for rapid parallel real-time system development . . . 23

1.3.5 The Otter parallel compiler . . . . . . . . . . . . . . . . . . . . . 23

1.3.6 CONLAB an interactive parallel MATLAB like environment . . 25

1.3.7 From MATLAB to a system-on-a-chip . . . . . . . . . . . . . . . 25

1.3.8 Slice hoisting for telescoping languages . . . . . . . . . . . . . . 27
2 Method

2.1 C-code test environment ........................................... 29
2.2 Generation of C reference code ..................................... 31
2.3 Translation to C target code ........................................ 32
2.4 Floating-to-fixed-point conversion .................................. 32

3 Benchmark Code

3.1 Hand coded floating point test programs ......................... 34
3.2 IEEE 802.16d WiMAX transmitter reference code ............... 34
3.3 Fixed-point code .................................................. 35
   3.3.1 The squared average of a vector ............................. 35
   3.3.2 WiMAX modulator ........................................... 35
   3.3.3 Digital filter in fixed-point ................................ 36

4 Compiler Description

4.1 Catalytic Tools ................................................... 37
4.2 Embedded MATLAB .............................................. 40

5 Floating-Point Code Generation

5.1 Results of compiler optimization ................................... 42
   5.1.1 Reference code .............................................. 44
   5.1.2 Target code ................................................ 45
   5.1.3 MEX and GCC–O3 ........................................... 46
   5.1.4 Conclusions to initial tests ................................ 47
5.2 Benchmarks on MATLAB intrinsics ................................ 47
   5.2.1 The function: sort ......................................... 48
   5.2.2 The function: fft .......................................... 50
   5.2.3 The function: conv2 ........................................ 50
   5.2.4 The function: xcorr ........................................ 51
   5.2.5 The functions: inv, mldivide, transpose, filter .......... 51
   5.2.6 The functions: mtimes, plus ................................ 52
   5.2.7 Discussion on results ....................................... 52
5.3 Tests on WiMAX compilation ....................................... 53
   5.3.1 Interleaving ................................................ 53
5.3.2 Convolutional encoding .......................................... 56
5.3.3 Transmitter burst ............................................... 57
5.4 Translation implications on WiMAX chain ......................... 58
  5.4.1 Catalytic Tools .................................................. 59
  5.4.2 Embedded MATLAB ............................................. 59
5.5 Conclusions on WiMAX ............................................. 60
  5.5.1 The compiler MCS ............................................... 60
  5.5.2 The compiler EMLC .............................................. 61

6 Fixed-Point Filter Implementation ..................................... 64
  6.1 Manual fixed-point filter design .................................. 65
    6.1.1 Design specification ......................................... 66
    6.1.2 Partitioning and coefficient computation .................... 67
      6.1.2.1 Cascade filtering ........................................ 67
      6.1.2.2 Parallel filtering ........................................ 68
    6.1.3 Scaling ....................................................... 68
    6.1.4 Section ordering and coefficient scaling for the cascade filter 68
    6.1.5 The parallel filter coefficient scaling ....................... 71
  6.2 Results: fixed-point filter ...................................... 72
  6.3 Fixed-point design aids from Catalytic Tools .................... 73
    6.3.1 The squared average of a vector ......................... 75
    6.3.2 IEEE 802.16d modulator ..................................... 76
    6.3.3 Filter design facilitation .................................. 76
  6.4 The MathWorks target design tools and Embedded MATLAB ....... 77
    6.4.1 The squared average of a vector ......................... 78
    6.4.2 IEEE 802.16d modulator ..................................... 78
    6.4.3 Filter design facilitation .................................. 79
  6.5 Conclusions ..................................................... 81

7 From an Engineer’s Perspective ..................................... 83
  7.1 Learning curve ................................................... 83
  7.2 Ease of MATLAB to C translation and usability of generated code 85
  7.3 Future outline ................................................... 86
List of Figures

1.1  Column major to row major issue between MATLAB and C. . . . . . . 13
1.2  A common product design process at Ericsson AB. . . . . . . . . . . . 15
1.3  The application of automatically generated reference code. . . . . . . 16
1.4  The application of automatically generated target code. . . . . . . . . 17
4.1  A GUI coming with MCS illustrating types inferred. . . . . . . . . . . . 38
5.1  Sort-Search: Different levels of compiler optimization. . . . . . . . . . . 44
5.2  Speed up on MATLAB operators and built-in functions. . . . . . . . . 48
5.3  WiMAX interleaver and encoder translation. . . . . . . . . . . . . . . . 58
6.1  IEEE standard single-precision floating-point format. . . . . . . . . . . 64
6.2  An example, fixed-point filter design process. . . . . . . . . . . . . . . 66
6.3  A direct form II second order filter section. . . . . . . . . . . . . . . . 69
6.4  Eight order IIR filter partitioning. . . . . . . . . . . . . . . . . . . . . 70
6.5  Cascade SOS ordering. . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.6  Parallel filter SOS interconnection. . . . . . . . . . . . . . . . . . . . . 71
6.7  Fixed-point to floating-point filter comparison. . . . . . . . . . . . . . . 74
6.8  Catalytic Profiling Tool. . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.9  Catalytic Tools fixed-point filter. . . . . . . . . . . . . . . . . . . . . . 77
6.10 Fixed-point filter object C-code. . . . . . . . . . . . . . . . . . . . . . 80
# List of Tables

2.1 Platform utilized for research work .............................................. 30

5.1 Sort-Search: orig. MATLAB compared to hand translation .............. 43
5.2 Sort-Search: Results of compilation with MCS and EMLC ............... 43
5.3 Intrinsics: Comparison of code lines generated .......................... 49
5.4 WiMAX interleaver and encoder translation ................................. 57
5.5 Generated lines of code for WiMAX compilation ............................ 57
5.6 MCS, EMLC summary on WiMAX .................................................. 63

6.1 Filter norms for the cascade filter sections ................................. 71
6.2 Filter norms for the parallel filter sections ................................. 72
1

Introduction

In 1980 Cleve Moler taught numerical analysis at Stanford University. Jack Little was working for a consulting company at this time. A colleague of Jack was attending Cleve’s course. To facilitate his lectures, Cleve was using an interactive matrix calculator written by himself. This matrix calculation program, denoted as MATrix LABoratory (MATLAB), was written in Fortran and accessing a few subroutines from the Linear Algebra Package (LINPACK) and the EIgenSystem Package (EISPACK). Through his colleague, Jack became introduced to MATLAB. Together with another colleague Steve Bangert, Jack rewrote MATLAB in C to be deployed on the International Business Machines Corporation (IBM) Personal Computer (PC). In 1984, Cleve, Jack and Steve founded MathWorks. The first sold order of MATLAB, was 10 copies to the Massachusetts Institute of Technology (MIT) in the year 1985 [27]. Currently, MathWorks with Jack Little as Chief Executive Officer (CEO) and Cleve Moler as Chief Scientist, is an international business with more than 1,800 employees [22, 25].

The computer language C is widely used for system implementation and the realization of a variety of applications. MATLAB, a computer language, is used in application domains such as signal and image processing, amongst others, for interactive algorithm design. These algorithms developed in MATLAB, are then often manually translated to C for system deployment. In addition, product development cycles can be found, where an intermediate C algorithm reference is created between MATLAB algorithm design and target realization. Reference code is a “simple” translation of the MATLAB algorithm to C without specific target design. Especially for products of greater
complexity, this intermediate stage of reference code generation is applied to facilitate, test and verify target code deployment.

Automatic MATLAB to C translation is of relevance, either for reference code generation, or for direct compilation to target C-code. The availability of efficient MATLAB to C translators, could speed up product design significantly. Additionally, by having an iterative product development cycle in mind, the maintenance of at least one C source code, can be saved. An example saving of source code maintenance could be the automatic generation of algorithm reference code; where in case of algorithm modification just the MATLAB source needs to be manually edited. The main question that motivates the work in this thesis is: how “good” is MATLAB to C compilation already at this point? To answer this question, two promising commercial compilers, namely [MCS] and [EMLC] are studied.

1.1 Compilers

The mathematical constant $e = 2.71828182845904523536...$ constitutes the theoretical base for the information representation of least redundancy. The closest natural number to $e$ is 3. There are information systems based on ternary (base-3 number system) data representation, but eventually the binary (base-2 number system) format succeeded in the field of information technology. Base-2 can be realized with an electronic switch (transistor); on = 1, off = 0, or the other way round. An example of an eight binary digit (bit) number is 01101010. Sequences of bits can be defined as, representation of instructions. Such instructions can then be used to invoke a certain behavior on hardware, which is usually composed of electrical circuits. Basically, hardware able to decode and execute binary instructions is composed of a memory and a processor. The memory normally holds a list of instructions and data to be processed. A processor, as the term already indicates, loads these instructions and performs the according transformations on the data. Such a list of instructions, describing a certain computation of input to output data is denoted as a program.

First computer programs indeed were designed and implemented in sequences of 0’s and 1’s [33, chap. 1]. However, a binary representation of information is hard to
comprehend by humans. Thus, in order to facilitate program design and inter-team communication, engineers first had their program instructions in form of words such as for instance LOAD, ADD or JUMP on paper. In addition, the hexadecimal (base-16 number system) or octal (base-8 number system) numbers were applied, due to their direct mapping to binary numbers. The practice of manual translation of such instructions on paper to *machine code* (bits), has been named as to *assemble* the program. In order to automate the assembly process of computer programs, *assembly language* was defined. These programs written in assembly language were then automatically translated to machine code by a program, which is referred to as *assembler*. An assembler needs to have a corresponding language as input and produces as output machine code for only one hardware architecture. Such abstract representations of programs following a specific language, which could be understood by each engineer of a development team, facilitated inter-team communication.

Still, due to the fact that each program which was deployed to a different hardware architecture needed to be rewritten, the hardware dependency of assemblers was tedious. In addition, it was still not very convenient to realize programs by for instance explicitly stating each LOAD, ADD, STORE or JUMP operation to reach a certain computation goal. With the increasing performance of processors, the availability of bigger and cheaper memories and a simultaneous rise of the amount of different vendors, another level of computer language abstraction was required. In order to further simplify program generation and to additionally enable automatic program deployment on different target hardware, *high level* languages have been created. This new level of abstraction allowed the entering of mathematical computations in infix notation and a simultaneous application of decimal (base-10 number system) numbers e.g. $5 \times (6 - 7/3)$. Moreover conditional statements, in other words control structures such as “IF some condition THEN do this ELSE do that” could be issued directly. Data types were defined to hold and address a particular amount of data for computation in the computer memory. Such data types could then be declared as constant or variables for inclusion in any expression.
An example could be the word *integer* or *int*, representing 16-bit of data and declaring the variables *a* and *b*:

```c
int a = 0;
int b = 6;
a = 5 * (b - 7 / 3);
return a;
```

In addition, *functions* representing certain parts of a program could be declared for repetitive execution. To summarize, programs with an increasing similarity to common human comprehension rather than the need to follow detailed computer operations could be designed and at the same time these programs could be deployed automatically on different platforms.

Naturally the closer the computer language is to human communication, the higher is their level of abstraction and complexity to be translated to machine code. There are translators for high-level languages to other high-level languages and also translators for high-level languages to low-level languages, where the latter [33, chap. 1] is defined as compiler. To compile something is normally understood as putting things together to a whole. According to [7]:

A compiler was originally a program that ”compiled“ subroutines [a link-loader]. When in 1954 the combination ”algebraic compiler“ came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

John Backus at IBM created the first complete compiler with the computer language *Fortran* in 1957. Ever since compilers were first introduced, the design of high-level computer languages, compilers and their optimization has been a “hot topic” for IT-companies as well as computer scientists. Nowadays, certain kinds of high-level to another high-level language translations, involving complex compilation processes, are also referred to as compilation [29, 5]; in this respect the present thesis does not follow the more rigid definition of a compiler as given in [33, chap. 1].
1. Introduction

1.1.1 The compiler process

In order to understand the different steps that the process of compilation implies, it is important to get familiar with certain terms; a good description of these can be found in [33, chap. 1]:

Syntax: Definition of expressions, statements up to whole programs is part of an imperative computer language; their tokens (symbols) and how they have to be combined in order to express a certain meaning need to be exactly defined to compile the program. These distinct language phrases represent the syntax, which is often specified by means of a context-free grammar\(^1\).

Contextual constraints (static semantics): In order to translate a program to machine code correctly, there have to be rules to guarantee their correct processing by the compiler. Rules such as how types are inferred throughout variables after declaration and how their type correct use can be checked are an important part of a computer language. The correct formation of a language phrase often depends on its context.

Semantics: The definition of program semantics can be seen from various aspects. From the operational point of view, semantic determines the “meaning” of a certain identifier within a program. A good example is the differentiation between global and local variables. In this respect a semantic property of a particular variable is its scope. In other words, to which parts of the program the relevant variable is visible.

These three major properties, syntax, contextual constraints and semantics, high-level languages must have, clearly define how to apply a certain computer language. The compilation process, which follows program design, is then built on these rules. Modern compilers usually comprise a front-end and a back-end. The front-end is responsible for “reading” the program, which is normally stored as text in a file. After that, the compiler front-end needs to check the program for correct syntax and contextual

\(^1\)A term referring to language parsers[4].
1. Introduction

constraints. This is to understand the programs semantics to further create an abstract intermediate language representation of the program. At this point the compilers back-end takes over to create assembly code, which is finally translated to machine code by an assembler.

1.1.1.1 Compiler front-end

In order to load the symbols relevant for compilation from a file into memory, lexical analysis is applied. The lexical analysis is based on rules, e.g. in form of regular expressions [4], which are formulated to instruct the lexical analyzer what is a regular token, and what can be discarded. A regular token is defined through the syntax of a program. These regular tokens, stored in the order they have been appearing within the program, are then run through a process called parsing. A parser checks the program for correct syntax according to the language’s grammar rules. At the same time an Abstract Syntax Tree (AST) is generated comprising of different levels; such as expressions, statements and functions. During semantic analysis, a symbol table where the operational semantics of a certain identifier can be looked up is generated. To be exact, these semantics of certain entities are additionally needed to check the program for contextual constraints.

Each imperative computer language has a certain type system: some programming languages are strongly typed and some are weakly typed. Whether a language is strongly typed or weakly typed is determined by how accurately a size of a constant or variable has to be specified by the programmer. Depending on the type system, a compiler needs to make sure that the program does not have any type mismatches: such as for instance between a variable assigned to another variable. Type correctness of a certain program is verified by a type checker, using the semantics from the symbol table and the rules defined by the type system of the computer language at hand. If the program is correctly typed, a final generation of an intermediate code from the symbol table is carried out. The applied intermediate language can be seen as an abstract assembly language which is independent of any particular hardware. In addition, intermediate code is next to information for debugging, free of it’s source language. An intermediate language representation of a program allows two tasks: either the goal is machine code,
or another high-level language. The part of the compiler after this intermediate stage is referred to as back-end.

### 1.1.1.2 Compiler back-end

The goal of a compiler is, in most cases, to generate assembly code and further machine code for a specific hardware. There are compiler back-ends running over the whole program several times. Such iterative approaches are for instance to find out data dependencies in order to rearrange code for faster execution while keeping functional correctness. In what way programs can be optimized depends also very much on the target hardware. Finally, as previously discussed, a “simple” process of assembling the program takes place. As a result, object code is often generated first, which just has to be concatenated by a linker, to form an executable file.

The description within the current section was a brief overview of a common compilation process. For modern state-of-the-art compilers each step described for the front-end and the back-end implies many optimizations by applying complex programming concepts; the reader is referred to [4] for more a detailed discussion of such aspects. The remainder of the current thesis will treat high-level to high-level language compilation; where next to efficiency, functional correctness, syntactical and contextual constraints, the readability for the humans can be an important factor.

### 1.1.2 Translation of MATLAB to C in contrast to compilation of C to Assembly

As stated in section 1.1 high-level languages facilitate programming by being more comprehensible for humans and less dependent on the target hardware. This goal is basically achieved by hiding details of the target hardware through applying different levels of abstraction. A drawback is that through high-level languages the programmer looses control over specific computer operations; as a result program design becomes dependent on the quality of the computer language and the compiler at hand. Translation from assembly code to machine code was straightforward because the information
content of an assembly file compared to its machine language representation is one to one. Each assembly instruction for a certain hardware can be replaced by its machine code representation. The high-level language C, as discussed in section 1.1.1, implies a complex translation process for code deployment on a target machine. In addition, a separate C compiler for each hardware architecture is needed. The compiler’s task is to automatically annotate the information needed by a specific machine to execute the program, which is not given through the language C.

MATLAB provides an even higher level of abstraction. C does not contain detailed hardware execution information, e.g. what needs to be loaded in what register at a specific point of the program. In contrast to MATLAB, C for instance still needs size information about constants, variables and data structures. Earlier, as C compilers had not been as well developed as they are now, assembly code often had to be optimized by hand. Still, C to assembly compilation was interesting, since a basic program outline was created. Often code parts that were tedious to write by hand in assembly language were equally adequately generated by the C compiler. Today, this approach is mostly limited to fields where highly optimized programs are needed, such as for embedded systems.

Hardware vendors targeting the domains of MATLAB, i.e. signal processing, often deliver C compilers with their products, with the result that MATLAB to C translation is becoming attractive. Embedded and digital signal processors may require C code, following specific hardware requirements. A need for such specific C-code links to the development of MATLAB compiler’s which take hardware information into account. Another reason for MATLAB to C translation is, as mentioned at the start of this chapter, generation of C reference code. Further, having C as intermediate language rather than assembly supports the manual application of optimizations. MATLAB to C can somehow be compared to going from C to assembly. The C compiler needs to automatically infer and annotate information about the hardware, in order to generate assembly code. From this perspective, the MATLAB compiler also needs to automatically derive and attach missing information. Such necessary information, e.g. type, shape and size of variables, enables the generation of C code. Due to MATLAB’s interactive nature, mastering automatic C-code generation is a challenge for compiler engineers.


1. Introduction

1.2 Problem definition

As discussed at the start of this chapter, MATLAB was developed as an interactive design environment, rather than for C-code generation. It is already a challenge to generate efficient C compilers, but in contrast to MATLAB, C was designed for compilation. Jeff Bier of technology analyst firm BDTI\(^1\) says about MATLAB to C translation in [16]:

This is a long-awaited development that will be welcomed by many, but it isn’t a silver bullet. The considerations that engineers focus on when they do algorithm development are different from the key considerations in embedded software or hardware development. For example, a code translation tool isn’t going to figure out how to segment your data and schedule your computations for efficient implementation, nor is it going to figure out where to apply parallelism.

The last statement of the quote from Jeff Bier in [16] may not be entirely true since there are already projects going on with the aim of parallelizing MATLAB. Examples of such attempts are given in sections 1.3.4 to 1.3.7. Due to the fact that MATLAB is an interpreted language, the understanding of interpreters to a certain level is important to understand the implications faced by automatically translating MATLAB to C.

1.2.1 Compiler versus Interpreter

By translating a particular source program in order to apply the syntax and semantics of a computer language, both compilers and interpreters are denoted as language processors. As already mentioned in section 1.1 a compiler is a special form of translator; interpreters, on the other hand, are more like a software processing hardware (machine) implemented in software. A special form of interpreters are emulators, which serve as an abstract machine. Emulators are for instance applied to test a certain machine design before it is actually produced; or to run more than one operating system simultaneously on one hardware platform.

---

\(^1\)Berkeley Design Technology, Inc.
The term interpreter indicates that the program will not be translated to machine code when executed; rather the code at hand is interpreted for a particular hardware platform. Consequently, the interpreter serves as a layer between actual hardware and the program. Good reasons for interpretation are [33, chap. 8]:

- Interactive program development, where the engineer is enabled to see the direct outcome of an instruction before entering the next since the need for compilation is not given.
- So-called throw away programs only executed once then discarded; where execution performance is not important.
- Programs expected to not run their instructions very frequently.
- In case of platform independent programs are needed.

Two types of interpretation are differentiated. Iterative interpretation follows exactly the same procedures as those that come into application on a real hardware machine. Fetch-Analyze-Execute are performed iteratively in order to execute the program, where the only difference to hardware is the execution time. This more straightforward behavior of iterative interpretation is applicable only if the source language is not a high-level language, or if it is pre-compiled to an intermediate language such as Java byte-code.

The second kind is recursive interpretation. To finally execute the program, concepts known from a compiler’s front-end come into application; these concepts are as described in section [1.1.1] verification of syntax and contextual constraints as well as the understanding of the program’s semantics. In the MATLAB case, recursive interpretation needs to be carried out. Information required for execution but not contained in the MATLAB code, such as type information, will be gathered by means of runtime context during interpretation. Interactive programming can only be realized through interpretation, but compiling a language created for this efficient way of algorithm design is problematic.
1.2.2 MATLAB to C synthesis issues

Cleve Moler and later the Mathworks developed MATrix LAboratory (MATLAB) with the array as basic-data type to enable a as natural representation of matrices as possible [27] [13]. Besides ordinary n by m matrices, MATLAB introduces four special matrix types: 1) a 1 by 1 matrix representing a scalar; 2) a 1 by n matrix representing a row-vector; 3) a n by 1 matrix representing a column-vector; 4) the empty matrix with the symbol [], cf. [15]. The empty matrix comes mainly into use when it is desired to shrink a matrix during program execution. Also, there is support for sparse matrices and functions for the corresponding operations in MATLAB. Ordinary matrices as well as the special matrices from number 1 to 3 can be represented with arrays in C. However, for the empty matrix and sparse matrices nothing comparable can be found in C. The matrix manipulations supported by MATLAB are similar to late versions of Fortran. This means, MATLAB operators and functions are overloaded to be applied on whole matrices. In contrast, C only supports elementwise operations on its arrays representing matrices.

MATLAB has a set of different data structures like cell arrays, java classes and structures. The language C also contains structures, but nothing similar to cell arrays or classes, which are part of object oriented languages [4]. The most commonly used data-types in MATLAB are real and complex. In C the data-type real can be directly mapped to the C data-type double. However, American National Standards Institute (ANSI) C does not contain a complex data-type and requires in this respect a workaround for MATLAB to C compilation. In terms of control structures and expressions, MATLAB resembles the imperative language C quite closely, with statements like for, if and case.

MATLAB has been developed into an interactive language, executed by means of an interpreter. This computer language allows the use of variables without any previous type declaration; even during program execution the types of variables can change. This means that MATLAB is weakly typed. Another interactive attribute is the dynamic change of size and shape of arrays that are possible and common practice in MATLAB. Due to MATLAB’s weak type system and the fact that variables of languages closer
to target architectures have to be precisely declared, type inference of MATLAB data elements is one of the main challenges for compiler engineers.

Next to its interactive attribute, MATLAB’s “strength” for algorithm development comes from its operator overloading, a set of Toolboxes\textsuperscript{1} and functions. As a result, it can deal with vectors and matrices in a natural way. Well-known algebraic problems such as the calculation of eigenvalues or the inverse of a matrix can be solved just by calling the corresponding built-in function. Additionally, MATLAB functions can be constructed with unlimited input and output parameters. For instance, one parameter could be a complex matrix passed by value. C for example can only pass a matrix with a pointer to the actual memory location of the matrix data (passed by reference).

MATLAB stores matrices column after column (column-major), not like C row after row (row major) in the computer memory. Figure 1.1 illustrates this difference between MATLAB and C. If automatically generated code follows the MATLAB memory layout, execution overhead for data transposing or in-place transposition of external data can be necessary. In addition, if real-time constraints are given, e.g. to wait until an image has been completely received to be able to transpose it, generated C-code following the MATLAB memory layout can be problematic. The automatic translation of MATLAB to “row-major” C-code is an additional challenge for MATLAB compiler designers.

There are three different kinds of MATLAB functions: 1) built-in or intrinsic functions; 2) M-files; 3) Matlab EXchange (MEX) files. M-files holding either MATLAB scripts or functions are the common way to store and call user-developed algorithms. Through a special interface construct, MEX files enable to call functions written in languages such as C or Fortran from the MATLAB prompt. As a result, simulations can be sped up through compilation, by simultaneously staying in the MATLAB environment. In addition, the use of MEX files is common practice for the verification of C or Fortran code using convenient MATLAB test benches.

Depending not so much on how these functions are stored, but more on the order in which MATLAB calls them, ambiguities may arise. Consider, for instance the statement $y = 1 + 2 * j$, if no user-defined function exists or no variable $j$ is defined, the

\textsuperscript{1}A term used by MathWorks for packages of functions.
1. Introduction

Figure 1.1: *Column major to row major issue between MATLAB and C.* The upper part of the image shows the declaration of exactly the same matrix for MATLAB and C. It can be seen that C starts array indexing with 0 and MATLAB starts with 1 to index arrays. The lower part of the image illustrates how the same matrix is stored in the computer memory by MATLAB and C.

An intrinsic function \( j \) is called, resulting in the complex variable \( y = 1 + 2i \). If, on the other hand, there is a \( j.m \) file in the user path holding the function \( j \) which simply returns 5 without input parameters, the above statement becomes \( y = 11 \). Further, if the MATLAB workspace holds \( j = 7 \), the consequence would be \( y = 15 \). Due to the fact that MATLAB is not intended for compilation, its interpreter needs to have a policy addressing the issue of how to resolve symbols. First, the interpreter’s dynamic symbol table is used to determine if there is a case of a variable. After that, the current directory is checked for user functions. Finally, if there is also no intrinsic or built-in function resolving the particular symbol, an error is produced.

The creation of a variable holding MATLAB’s base data element, the matrix, can be done in many ways. Three of these ways are more common than others; one approach is to use a function like \( C = \text{ones}(4,7) \) creating a matrix of size \( 4 \times 7 \). A second way is to initialize the matrix by directly typing the values for each position like \( b = [3 \ast \log(3) \ 1:0.5:4] \), where functions or a construct can be used to create values of a particular range and interval. Matrices can also be created by the use of a subscript for example \( A(2, 3) = 13 \), which can create a \( 2 \times 3 \) matrix holding zero on each position, except the indexed one. The value of a certain position of the matrix can be changed...
trough subscripting naturally, but there are also ways to do this collectively; for instance
A(:) = 3 addresses the whole matrix and assigns 3 to each element, B(1,:) displays all
the elements of the first row of matrix B and C(:,4:−1:2) prints out the columns of
matrix C from 2 to 4 in reverse order.

The constructs mentioned in the previous paragraph allow a quite efficient translation
to another high-level language, but it runs into problems regarding the dynamic resizing
of data elements. Such resizing happens easily in MATLAB by for instance assigning
a value to a subscript of a matrix exceeding its boundaries. It becomes even more
difficult, when the final size of an element depends on a runtime variable, such as for
instance dynamic resizing through control structures like a for-loop:

\begin{verbatim}
function y = loop(x)
    for i = 1:x
        y(i) = x;
    end
\end{verbatim}

In such cases the size of the return argument of the function \textit{loop} depends on the value of
its input argument. Further functions like the \textit{square root} or \textit{logarithmic} functions can
produce complex values when the corresponding variable becomes smaller than 0 during
runtime. Such a case could for instance be the following function call: \textit{sqrt}(b − c);
when \(c\) becomes bigger than \(b\) during runtime, MATLAB generates a complex value.
However, in C a statement like \textit{a = sqrt}(-1), results in a compilation error.

\subsection{Scope of investigation}

The idea to evaluate automated MATLAB to C translation was initiated by Ericsson AB\textsuperscript{1}. This company applies MATLAB for signal processing algorithm development and simulation. Furthermore, the MATLAB-code (\textit{M-code}) created is utilized to design, test and verify implementations in C as well as in Hardware Description Language (\textit{HDL}). In addition, due to the greater complexity of product development cycles at Ericsson, the generation of intermediate C algorithm reference code is applied.

\textsuperscript{1}A global telecommunications equipment supplier.
Due to a pre-study and discussions between Ericsson AB, Halmstad University and the author of this thesis, three major aspects for investigation on MATLAB to C translation were defined: 1) generation of C reference code; 2) translation to C target code; 3) floating-to-fixed-point conversion.

**Reference code:** Application reference code should be generated from any relevant MATLAB algorithm, involving minimal additional effort. Performance is not a major criteria, however, for bigger simulation and verification runs of interest. Functional correctness and accuracy are crucial. The application of automatic reference code generation is visualized in figure [1.3](#).

**Target code:** Performance measures, such as execution time and memory consumption, as well as the suitability to be deployed on embedded targets, are of major importance. The MATLAB language should be supported as well as possible, in order not to replace the overhead of manual MATLAB to C translation with manual MATLAB to “MATLAB- compilable” translation. In addition, in order to be
regarded as valid translation, functional correctness and accuracy are required. Automatic target code generation applied to a possible product development cycle is illustrated in figure 1.4.

**Fixed-point code**: Tools provided to address floating-to-fixed-point conversion through compilers at hand are studied. From this perspective, the abilities of corresponding compiler add-ons to facilitate an engineer’s fixed-point design approach is of interest. The achievable precision through automatic translation is studied. An introduction about floating-to-fixed-point conversion is given in the beginning of chapter 6.
Two compilers available as commercial products are chosen to demonstrate the state-of-the art in automated generation of C-code from MATLAB. In fact, both software tools are already in use at certain industries \[10, 16\], which supports the choice to investigate these:

**Matlab to C Synthesis (MCS):** Launched December, 2006 by Catalytic Inc., California, USA. More information about MCS can be found in section 4.1.

**Embedded MATLAB C (EMLC):** Brought to the market by MathWorks, Massachusetts, USA, together with MathWorks’ launch of MATLAB 2007b in August, 2007. In section 4.2 EMLC is described in detail.
1. Introduction

1.3 Related Work

1.3.1 MCC

The MATLAB C Compiler (MCC) developed by the MathWorks translates from MATLAB code to C/C++ source code so that it can be compiled on the desired platform. This software also translates display functions using C/C++ libraries; including a C/C++ math library as part of the compiler package [21]. MCC does not aim to generate optimized translations; rather, it functions well for protecting proprietary source code as well as for creating reference applications, such as simulations. By utilizing the generic data-types rather than type inference algorithms, in which calls to libraries dealing with arbitrary argument types are mostly undertaken [12], the tool sacrifices big speed-up potential. As result in [12, 14], compared to other MATLAB to “target close language” translators, MCC pays the price for its generic characteristics by an unconvincing performance.

1.3.2 The FALCON project from De Rose

Luiz De Rose saw a big potential for an automated translation of MATLAB to a language closer to hardware. In fact it is worth quoting the following statements from [15]:

To implement scientific code a programmer could develop a prototype in an interactive language, like MATLAB, and then rewrite it in a compiled language, like C or Fortran. In practice, however, the overhead of re-implementing programs in a different language is large enough that most people seldom follow through with this option. Clearly the best solution is for programmers to use a translator that directly generates efficient code from MATLAB programs.

Today it has become common practice to first implement, simulate and verify in MATLAB by way use of its graphical display functions in addition to its interactive programming, and then go on to hand-translate to C code.
As already mentioned in section 1.2.2, the inference of the correct data type, the shape and size of a variable and taking the variable’s dynamical extendability dependent on runtime conditions into account, is a big challenge for translation tools. The two major methods used by Fast Array Language COmpilation (FALCON) in order to translate from MATLAB to Fortran90 are on the one hand static inference generating declarations at compile time and on the other hand strategies that are applied to the translated code to resolve those cases that could not be inferred at compile time. The first stage is to create an intermediate language representation of the MATLAB program in question, which is called Static Single Assignment (SSA). As the term already indicates, each scalar variable in SSA is assigned to one statement at most and there is only one definition for each use of the variable. Since MATLAB is changing its binding of types during runtime, it is beneficial to see clearly where a particular variable has been used throughout the program. Another simplification of SSA is that it deals uniformly with arrays and scalars. In other words: a full array assignment is represented by one statement.

There are several ways for the compiler to search and infer the type, shape and size information of different variables. FALCON gives the possibility to provide input files of data to be used with the program in order to aid the static inference process. In the static inference process only type and shape information is extracted from the input files, due to the likelihood of dynamic size changes in MATLAB programs. Program constants serve the software tool to infer the type and shape as well as the size of variables. By considering the conformability requirements imposed by operators, shape and size information, and in the case of logical values, type information, can be gathered. When built-in functions are used, the type inference of output types is possible according to its input types. All the type, shape and size information that is gathered is stored by means of a database which also holds a result table. This information is then propagated forward in order to infer statically as many types as possible through the intermediate code. Due to the heavy overloading of MATLAB operators, backward propagation is of little help and therefore not executed [15].

The type inference that is hierarchically implemented in FALCON starts with the logical type, goes further to the integer, and is then followed by the real and finally
the complex data type [15]. The complex data type can represent all other types, but
composes the highest computation cost. The type of each variable is considered as
NULL from the start, at which point type inference is executed in loops. After each
loop-cycle the type of the specific variable is propagated one step higher in the type
hierarchy until its type is determined or marked as unknown. If ambiguously typed
expressions occur, such as for instance the square root of a variable, the type can be
solved through value propagation or it is declared as complex.

FALCON infers shape information through the propagating of operations on variables.
In this process the differentiation between row and column vector is crucial. In many
cases it is impossible for FALCON to determine array sizes, and these are therefore left
for dynamic analysis at run-time. If there are still unknown data-types left after static
inference, code is generated to differentiate between real and complex, as well as for
memory allocation during program execution. This approach is costly but still cheaper
than running the whole code with complex numbers for the particular variables. As
already mentioned, size inference during runtime is more likely to appear than dynamic
type inference in practice, in which case FALCON applies some optimization techniques
for the placement of memory reallocation code. However, this dynamic approach is
quite costly.

The benchmarking of mathematical algorithms in [15] showed that in most cases the
hand-coded version runs negligible faster than FALCON. Due to the better control
structure of the compiled code, element-wise operations, compiled by FALCON, could
achieve the biggest speed-up compared to the MATLAB interpreter, whereas algo-
rithms which made use of many built-in functions resulted in less speed-up [15]. In
case of the linear equation (MATLAB left divide) \( x = M \setminus (N \ast x + b) \), the hand-
coded version showed significantly more speed-up than FALCON, which was basically
due to its use of the library Basic Linear Algebra Subprograms (BLAS), rather than a
lack of inferred types. Compared to measurements where type inference has been de-
activated, code could be executed up to 25% faster by use of FALCON’s type inference
engine. In this, the importance of MATLAB type inference for the translation process
was evident. To sum up De Rose’s project illustrated the potential of MATLAB code
translations, and provided new ideas for further research.
1.3.3 MaJIC an alternative to the MATLAB interpreter

The compiler described in [3] does not aim to produce a target-close code for stand-alone compilation and execution. The concept of Matlab Just In Time Compiler (MaJIC) is to compile code as late as possible, in order to speed up the MATLAB code execution without sacrificing the interactive programming with MATLAB. As the term MaJIC already indicates, compilation happens during runtime, where the manner of compilation is the interesting aspect. There are two concepts facilitating MaJIC compilation; first a speculation algorithm looks ahead of the current program state and tries to find patterns to infer information about type, shape and size during runtime. If successful, optimized compilation of the particular part is executed. The second procedure takes over if speculation fails to gather enough information to pre-compile the particular part of the program. In this manner, a fast, but less optimized, just-in-time compiler is used to make the relevant part of the code executable.

Similar to FALCON but with a few improvements, the MATLAB code is scanned and then parsed to create an AST. Next, the AST is translated to a static symbol table holding no type information. As a third step, type inference is started, where the two different compilation modes come into account. In the speculative mode only the information from the AST and the symbol table is used to infer information about variables. The advantage of having complete information about the program context available at runtime, such as the MATLAB interpreter has, in that it simplifies the inference process when it comes to Just In Time (JIT) mode. The final step is either the fast built-in memory for JIT compilation and execution; or C or Fortran code is generated in the speculative mode, compiled and linked with platform-native tools and put in a code repository. During program execution, an interpreter will request a so-called type-matching-system for a semantically correctly compiled code to the given invocation. If the call is successful, execution continues in a platform optimized manner, if not, JIT mode is triggered.

For type inference processes different from FALCON both forward and backward type propagation come into use, which is executed by the speculation mode [3]. The forward mode is the propagation of type, size and shape information of input arguments over
the function body by use of a type calculation based on 250 rules, e.g. integer-scalar-multiply, complex-vector-multiply, real-matrix-multiply. The information inferred from this process is then annotated to the AST. Backward mode means that the type speculator attempts to infer information about input parameters through the function body. There are several rules on which the type speculator is based [3]:

- Indexing constructs to specify matrix ranges with the colon operator (:) are almost always integer scalars. This links to the fact that complex numbers are not supported and fractional numbers become rounded; e.g. $A(a:b, :) = 4$
- Relational operators also disregard complex numbers, and between vectors these operations are very rare, since they are non-intuitive.
- In case a variable known as scalar is used within brackets as part of a vector, mostly all of the other variables that construct the vector will be scalar; e.g. $[a1 \ a2 \ a3]$
- When variables serve as a subscript index like $A(\text{idx}, \text{idy})$ or are part of an expression serving as subscript, they are likely to be scalar.
- Functions to create matrices such as e.g. rand(), ones() and zeros(), mainly have integers as input arguments.

As a complement to forward propagation the speculation executes backward propagation in loops until type convergence has been reached. After this the corresponding part of the program is compiled to a temporary file with the most aggressive compilation mode available at the platform of execution [3]. This compilation process can take several seconds, but creates executables running much faster than the JIT compiler. The benchmarks run in [3] which show that type speculation tends to succeed where it is most needed. However, it can fail in two ways. When speculation is too aggressive useless code is generated. For instance, too many versions of the same function are created. As a consequence of not being sufficiently aggressive, suboptimal code is generated. Type speculation is an interesting concept since it deals with the probability that certain constructs are used by programmers.
1.3.4 RTExpress for rapid parallel real-time system development

The world of computers is becoming increasingly parallelized. However, concurrent programming remains a hard and error-prone task. Consequently, projects such as those described in [8] were launched to provide solutions enabling rapid algorithm development in MATLAB, by way of automatically translating these programs to parallel High Performance Computers (HPC) “native languages”. Real-Time Express (RTExpress) is a concept to enable even algorithm developers without much experience as concurrent programmers to deploy their code efficiently on parallel computers. By use of this tool the functional decomposition and selection of target architecture should be enough to create high performance parallel code. For this reason a so-called target balancing tool is used to partition the MATLAB M-files into groups and further into instances of groups. This approach results in the compilation and execution of parallel code in Single Instruction Multiple Data (SIMD) manner. Unfortunately for this project MCC has been chosen as compiler. As already discussed in section 1.3.1 MCC is a compiler with a moderate translation performance. The resulting C-code is then post-processed, compiled and linked with the optimized native target compiler. This is also the point where powerful libraries like the Scalable Linear Algebra PACKage (ScaLAPACK), which holds highly parallelized algorithms to compute linear algebra problems, or the Message Passing Interface (MPI) for inter process communication, are added. Benchmarks in [8] have shown great potential for these kinds of solutions.

1.3.5 The Otter parallel compiler

Outlined in [29], similar to RTExpress Otter translates from MATLAB to parallel SIMD style C code. The use of a compilation process similar to FALCON rather than MCC is one of the differences.
Several steps are run through inserting optimizations by the so-called multi-pass compiler [30]:

1. As for most other compilers, an AST is first produced through scanning and parsing.

2. The creation of the intermediate language representation SSA such as it came into use for the FALCON project, is the next step [13, 14].

3. Next, type, size and shape inference, as far as achievable at compile time, is undertaken; As for FALCON, run-time dependent information about variables, is solved through corresponding code generation in this step.

4. Step four modifies the AST to shift terms and sub-expressions, which require inter-process communication, to statement level. As a result these constructs can be translated to call the run-time library.

5. Since the program is translated in SIMD manner, the next step is to assign the particular parts of the program to the correct process. Here the relevant statements are surrounded by conditionals. Library functions for communication purposes are inserted as well.

6. As a sixth step, a process denoted as peephole optimization is carried out. This step involves that the compiler attempts to detect calls to the run-time library, which can be combined to a single call.

7. Finally the AST is traversed and C code is generated.

The runtime library is an essential part of the Otter compiler. It uses the ScaLAPACK mathematics library, which is also implemented at the RTExpress project, and is a parallel version of Linear Algebra PACKage (successor of LINPACK) (LaPACK). The LaPACK library is a successor to LINPACK, the co-developer of which was Cleve Moler, the inventor of MATLAB [27, 26]. As a consequence many computations in MATLAB have been based on LINPACK [20]. In addition, the parallel version of the Fastest Fourier Transform in the West (FFTW) library developed by the MIT and the MPI for inter-process communication are part of the Otter compiler.
The parallelization of ScaLAPACK is based on a logical grid layout of the different processors. The interesting aspect and part of the investigation in [30], is that the way in which this grid is laid out influences execution performance. For instance, matrix multiplication is most efficient if applied on a square grid layout. In the case of matrix-vector multiplication and the calculation of maximum or mean run fastest in a row grid. A column shaped grid is the optimum for calculations of the minimum. The following ideas for future work have been drawn from various benchmarks in [30]. Since it is often possible to determine how frequent a function will be called already at compile time, and parallel performance is significantly impacted by data distribution; potential has shown up for automating compilers for parallel MATLAB translation to undertake decisions on data distribution.

1.3.6 CONLAB an interactive parallel MATLAB like environment

As opposed to RTExpress and the Otter compiler, Concurrent Laboratory (CONLAB) uses the Multiple Instruction Multiple Data (MIMD) technique to execute programs either in distributed or shared memory architectures [32]. Umeå University in Sweden implemented a research environment, which does not intend to translate to any other computer language. CONLAB is rather a subset of MATLAB enabling interactive parallel execution without compiling and linking. Partitioning information is expressed in relevant MATLAB scripts like it would be a simple control structure. The arguments given to this partitioning statement are used for initializations as well as for process assignment to virtual processors. By using MATLAB as programming language and reducing partitioning and architecture simulation overhead, this interactive solution enables fast research in the field of parallelism.

1.3.7 From MATLAB to a system-on-a-chip

As the term MATLAB Compiler for Heterogeneous systems (MATCH) already indicates, the compiler described in [5] is designed to partition and generate code for
computer systems comprising of different architectures. In fact MATCH is able to translate MATLAB code to C code for embedded processors as well as for digital signal processors and Central Processing Unit (CPU)s. But the even more interesting part of this translator is its ability to create HDL code for Field Programmable Gate Array (FPGA) and specialized chip production. As a demonstration environment the team around MATCH has connected a Xilinx™ FPGA board, a Transtech™ Digital Signal Processing (DSP) board, a Motorola™ embedded processor board and a Sun™ CPU via a Versa Module Europa (VME) bus and Ethernet; visualizations and detailed specifications can be found in [5]. The compiler maps automatically suitable functions to the targets attributes. For instance some operations require floating point computations, which are not suitable for implementation on FPGAs. Moreover, experienced programmers can “fine-tune” code generation by providing detailed directives about the hardware specification to the MATCH compiler. The compilation process is similar to the Otter compiler:

1. First the MATLAB code is scanned parsed and an AST is generated.

2. This is followed by several phases involving modifying and annotating of the AST. In this process the compiler is much stricter than solutions like FALCON or Otter. User code annotations in the form of %!match followed by type, shape and size information are required to resolve certain inference problems. Expensive runtime checks can be saved applying this approach.

3. Next, data and control flow analysis is used together with programmer-directives about hardware to partition the AST into corresponding sub-trees.

4. As known from the Otter compiler MATCH maps also library functions onto respective targets, where procedural code is encapsulated to user-defined procedures; but the big advantage is that MATCH is also able to deploy certain procedures to FPGAs by creating corresponding Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) code.

5. Finally the main thread of control is generated for the Sun™ CPU as required for the SIMD parallelization technique. This main thread of control performs remote procedure calls to the nodes executing the equivalent function.
In order to implement the suitable MATLAB functions to FPGAs, the VHDL code from MATCH, which is generated at Register Transfer Level (RTL), can be used by common synthesis tools from the industry. Common procedures that are mapped on FPGAs are matrix-multiplication or addition, one dimensional Fast Fourier Transform (FFT) and filter functions. Control structures in MATLAB code are translated to their equivalent VHDL representation. To represent assignments in VHDL variables are used. In order to implement loops to FPGAs in a safe manner, a finite-state-machine with four states is used. State 1 performs initialization of loop and loop-body variables and state 2 checks if the loop exit condition is satisfied. If yes, the next step will be state 4 which represents the loop exit. If no, state 3 will be used to execute the loop body. In case there are operations to be carried out, such as read or write from or to memory, more states will be integrated. Procedures which are mapped on embedded or digital signal processors will use MPI for communication; but due to the limited computation power and memory of FPGAs, basic communication functions are required for chip creation.

Test runs on the previously mentioned setup in [5] show great potential for this kind of automatic translation from MATLAB to heterogeneous computer systems. A next step could be a compiler for automatically deploying programs developed in MATLAB to a system-on-a-chip.

1.3.8 Slice hoisting for telescoping languages

The word “telescoping” has evolved from the approach of extending a computer language in an hierarchical manner by repeating the process of library building as presented in [13]. The interesting part of telescoping languages is a technique to resolve runtime dependent resizing of variables at compile time. Whereas the FALCON compiler is only able to generate code to deal with this problem at runtime, slice-hoisting is often able to infer this information through code transformations [12] already at compile time. A first step of telescoping languages is to gather type, shape and size information in a static manner similar to De Rose’s work, but in a slightly improved way by applying backward propagation [15, 12]. Still, array sizes defined through subscriptions comprised of expressions, that change their value during execution, or dynamic
resizing dependent on the runtime values of variables, can not be inferred statically. In previous projects, this issue usually resulted in code performing expensive resizing during runtime. The key is to pre-allocate the array once it has reached its maximum size required throughout the program.

Slice-hoisting identifies the code responsible for the resizing of the array; it takes the particular slice and hoists it before the first use of the relevant variable, as illustrated in [12]. Consequently, the maximum size of the variable can be determined before it is used the first time. This enables an allocation of the maximum memory required in one operation, rather than continuously having to reallocate space. The technique of slice-hoisting can be applied in two very common cases:

- When the array size changes due to index expressions.
- In the cases when array sizes involve symbolic values dependent on runtime conditions.

There are still cases where slice-hoisting can not be applied, but as illustrated in [12], quite a few of the variables of different DSP algorithms could be inferred during compile time with this technique.
Method

The problem definition given in section 1.2.3 suggests three major aspects for further study: 1) generation of C reference code, 2) translation to C target code, 3) floating-to-fixed-point conversion. Thus, in order to address these problems, the method applied in the present thesis consists of three parts. There are several ways to investigate compilers. For instance, one way is to focus on a small set of algorithms and then apply these on many different relevant platforms. Another way is to concentrate on one platform and cover a larger set of algorithms. The goal of the present thesis is to provide a broad view about MATLAB to C translation within the given time resource, rather than to concentrate on how certain translations perform on different target platforms. This motivates the choice of having only a generic Linux computer as platform which is illustrated in table 2.1.

2.1 C-code test environment

The C-code displayed in appendix A, referred to as driver, is used to address the floating-point benchmarks contained in the present thesis. However, minor changes are applied to the program to suit a particular case. This is required since many different data-types and shapes for the different investigations are to be supported. The driver contains a main function, which loads test-data from a text file created with MATLAB. Consequently, it can be assured that the same test vector is used for interpretation in MATLAB as well as for execution on the target processor. Memory
Table 2.1: Platform utilized for research work.

<table>
<thead>
<tr>
<th>Platform Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU:</td>
<td>Intel® Pentium M, 2 Mega Byte (MB) cache, frequency set to 1 Giga Hertz (GHz), single core</td>
</tr>
<tr>
<td>Front Side Bus (FSB):</td>
<td>333 Mega Hertz (MHz)</td>
</tr>
<tr>
<td>Random Access Memory (RAM):</td>
<td>1024 MB</td>
</tr>
<tr>
<td>M-code Compiler 1:</td>
<td>MCS 2.0-2252 on MATLAB 2007a</td>
</tr>
<tr>
<td>M-code Compiler 2:</td>
<td>EMLC on MATLAB 2007b</td>
</tr>
<tr>
<td>C-code Compiler:</td>
<td>GNU Compiler Collection (GCC) 4.2.2</td>
</tr>
<tr>
<td>Target Operating System (OS):</td>
<td>i686, PC, Linux, GNU’s Not Unix (GNU)</td>
</tr>
</tbody>
</table>

Allocation for the test data is carried out dynamically, to ensure convenient use of the driver for many different benchmarks. In case the return data from the tested function is something else than one scalar, memory is allocated for these data as well. Finally, all dynamically allocated memory is freed.

To measure execution time the call of the generated top-level C-function is wrapped with a construct of the intrinsic C-function `gettimeofday`. This function, as opposed to the function `clock`, returns the OS system time and not the number of processor clock cycles since program start-up. This approach proofs to be more applicable than the C `clock` function. Consequently, in order to generate meaningful test results, the benchmark CPU’s frequency has been stabilized to 1 GHz. Additionally, the running processes on the test system have been viewed and minimized. Nevertheless, small actives of a multi-tasking OS such as Linux, can influence the benchmark results notably, if the current task is not of considerably larger size. This issue is addressed through the design of test data invoking benchmark times around 0.01 to 10 seconds.

Each test run is executed at least ten times and checked visually to avoid fluctuating measurement of the execution times. One value out of these ten values generated per test run would provide a sufficient accurate result for conclusions to be drawn in the present thesis.
This finding is due to:

- The fluctuation of measured times has to be negligible small for a benchmark to be considered meaningful.
- Goal of the current thesis is to provide an overview of MATLAB to C compilation quality and problems, rather to give detailed performance information of selected algorithms.

However, three out of ten values generated are copied into a spreadsheet to serve as minimum evidence to have regular test results gathered. Finally, the average of these three values is computed and regarded as test result. In addition, randomly chosen benchmarks are undertaken again after a computer reboot, to assure that executed benchmarks can be repeated. The verification of functional correctness and accuracy is executed in two steps. Step one is the generation of a \texttt{MEX} file to directly compare the results in MATLAB. Step two is a print-out of test results in the Linux shell. This print out is then copied into MATLAB, to be verified with the results from the corresponding interpreted \texttt{M-code}. In some cases a collective value, such as \texttt{sum} or \texttt{norm}, is computed, printed out and visually verified with the MATLAB result. The computation of the speed-up applied in the present thesis follows: $Sp = \frac{T_i}{T_c}$ with $T_i$ as the execution time of the MATLAB 2007b interpreter and $T_c$ as the execution time of the corresponding C-program.

## 2.2 Method one: generation of C reference code

The translation to reference code described in section 1.2.3 is investigated by means of method one. These studies utilize the test environment described in section 2.1. Due to the fact that execution time is not the primary objective, the GCC compiler shown in table 2.1 is used without any flag. The documentation of the compilers mentioned in section 1.2.3 is studied thoroughly, to assure application of minimal effort applied to invoke reference code generation. Finally minimal annotations to the \texttt{M-code} are made to enable compilation and further evaluation of the resulting C-code.

\footnote{Description of MEX can be found in section 1.2.2}
2. Method

2.3 Method two: translation to C target code

Method two is defined to investigate on the problem of automated compilation of MATLAB to C target code described in section 1.2.3. Similar to method one the approach described in section 2.1 is applied, to examine target code generation. GCC is used with compiler optimization flag `-O3` in order to evaluate the speed-up achievable compared to the MATLAB interpreter. Often speed-up can be improved through the use of more memory, where it comes to a memory to execution time tradeoff. In this respect, the challenge for target program designers is to balance memory usage and execution time, to fit application requirements and target hardware best. Suitability for target implementation of generated C-code is evaluated through the inspection of the according C-files. In this way, aspects such as the dynamic memory allocation utilized (yes or no) and the amount of memory required are addressed: where the latter is estimated.

Initially, optimization possibilities offered by the compilers which are subject of study are explored. Tutorials, documentation and small test programs are used to assure a necessary level of comprehension. These optimizations found in 23 9 11 will be applied step by step, the result is verified after each test run. In contrast to the generation of reference code, where the least effort required to invoke a translation is of interest, M-code optimizations for the particular compilers used in this study is of importance for code generation. From this point of view the balance between the production of efficient C-code and MATLAB language support, is studied. Also so-called “soft-values” such as generated lines of code and time roughly required for M-code manipulations are taken into account.

2.4 Method three: floating-to-fixed-point conversion

The problem of fixed-point code generation as defined in section 1.2.3 is studied through the application of method three. The platform as given in table 2.1 is utilized for
method three. MATLAB is used in the signal processing field requiring C-code to be implemented in fixed-point arithmetic. The compilers introduced in section 1.2.3 offer different tools facilitating fixed-point simulation and C-code generation from floating-point M-code. In order to evaluate possible design processes and the efficiency of tools available in a meaningful way, a manual implementation of a digital filter in fixed-point is undertaken. As a next step, comparison of design efficiency gain by application of given tools is undertaken. In this respect the tools are first benchmarked on the somewhat smaller problems described in sections 3.3.1 and 3.3.2. The subject of measurement is in the first place the Signal to quantisation Noise Ratio (SNR) in deciBel (dB) compared to the interpreted floating-point MATLAB filter function. Equation (2.1) illustrates the SNR computation applied, where vector \( v \) is the floating-point reference and vector \( y \) holds the fixed-point results.

\[
\text{SNR} = 20 \cdot \log_{10} \frac{\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}}{\sqrt{(y_1 - v_1)^2 + (y_2 - v_2)^2 + \ldots + (y_n - v_n)^2}}
\]  

(2.1)

The generated code is interfaced through MEX with the MATLAB prompt in order to compute the SNR. As target hardware for all fixed-point simulations a 16-bit processor with 32-bit accumulators is assumed.
3

Benchmark Code

The methods one and two described in sections 2.2 and 2.3 are applied on the benchmark code described in sections 3.1 and 3.2. The benchmark code described in section 3.3 is used with method three described in section 2.4.

3.1 Hand coded floating point test programs

In order to evaluate the translations of MATLAB intrinsics and frequently used toolbox functions on different data types and data structures, small test programs are designed in M-code. In many cases a manual translation of M-code has been produced as well. In this way a direct performance measure between manual and automated translation can be generated. More details regarding code used with benchmark results for specific investigations is given in sections 5.1 and 5.2.

3.2 IEEE 802.16d WiMAX transmitter reference code

The second kind of code used for benchmarking is a baseband digital signal processing chain. Ericsson AB provided a WiMAX transmitter developed by one of its professionals in M-code by following IEEE standard 802.16d [1] ch. 8 described in [17].
Motivating aspects for the use of this code are:

- The code was developed by an experienced DSP programmer and therefore resembles programs found in the industry.
- MATLAB development has been made without the aim of automatically translating the program to C.
- For WiMAX, state-of-the-art DSP algorithms were applied. As a result the corresponding M-code holds relevant code constructs and functions to be supported by modern MATLAB to C compilers.
- Not just separate functions, but also the interoperability of certain parts of the chain can be tested.
- Some IEEE 802.16d functions require the smallest data type possible, by working on matrices holding binary data. On the other hand a number of functions are designed for floating point data types. In that way data type handling of the particular translation tools can be benchmarked on a more practical example.

### 3.3 Fixed-point code

#### 3.3.1 The squared average of a vector

This function takes a vector as input argument, squares each element and computes the sum. Finally the sum is divided by the length of the vector to provide the average value as output argument. Basic mathematical operations such as add, multiply and divide and some looping are used. Consequently, the method can be concentrated on basic fixed-point arithmetic application by a certain compiler. A manual translation to fixed-point C has been also produced, to evaluate accuracy of automatic translation.

#### 3.3.2 WiMAX modulator

The WiMAX modulator is part of the DSP chain described in section 3.2 and executes a transition from integer to complex floating-point types. WiMAX modulation serves as
a “simpler” practical example for floating-to-fixed-point conversion. Due to being free of any MATLAB toolbox function, the modulator serves as a next step of complexity to the code described in the previous section.

### 3.3.3 Digital filter in fixed-point

In chapter [6] the implementation of a digital filter in fixed-point is described. Following a whole manual filter design process, a fixed-point filter in C is realized. The object for floating-to-fixed-point translation is M-code that holds the digital signal processing toolbox function `filter`. This `filter` function is used with the same filter coefficients as for the manual design process. As a result the compilation outcome of the investigated translators can be compared to the manually designed code in a meaningful way.
Compiler Description

The two compilers described within this chapter are conduct of detailed investigation for the present thesis.

4.1 Catalytic Tools

Catalytic Tools is a software set developed by Catalytic Incorporation. This tool set is comprised of Rapid Matlab Simulator (RMS) and MCS. To speed-up MATLAB, RMS compiles M-code transparent to the user with C as intermediate stage. On the other hand MCS generates ANSI C-code, but can also like RMS generate MEX functions. In section 1.2.2 it is described that MEX conveniently enables the verification of generated C-code, using the MATLAB environment. In contrast to MCS, RMS can speed-up MATLAB simulation without having support for C translation of all functions contained by a particular algorithm. In order to generate “stand-alone” ANSI C-code, MCS needs to have support of all functions contained by a program matter of translation. According to [2], MCS keeps the intermediate representation of the program close to source level in the form of an AST during compilation. SSA or abstract assembly as intermediate language is not created; instead, control/data flow graphs and data dependence graphs are annotated to the AST.

In order to invoke translation MCS needs at least the input argument’s type and shape given as annotations of specific format to the M-code. The closer the type declaration meets the actual runtime context, the more optimized the C-code will be.
4. Compiler Description

Figure 4.1: A part of the GUI coming with MCS illustrating types inferred of a particular function; by permission from Catalytic Inc.

generated. In order to generate code to be used with different data types, size and shape, it is sufficient to use the most generic declaration. Such a universal declaration could for instance look for variable \( M \) like \( mbrealmatrix(M) \) (\( mb \) stands for must be). Optimization can be achieved by specifying type, size and shape accurately, which leads to a more constrained program. A declaration such as for instance \( mbintrow(M); mbsize([1 512], M) \), generates code operating with integers instead of doubles by statically allocating the data-array. Consequently, the resulting program can be significantly optimized, but is restricted to operate on a vector holding 512 elements and does not run on floating-point data types. These additional input argument declarations in the M-code can be seen as a workaround to the missing runtime context at compile time. Type inference is executed by MCS, so as to annotate the missing type, shape and size information of internal variables and the output argument. MCS comes with a Graphical User Interface (GUI) to visualize variables inferred according to a particular M-code input argument declaration. Variables such as for example in figure 4.1 can be clicked to highlight their corresponding representation in the M-code. This GUI can be useful to check the types inferred, before invoking C-code generation.

Another way to achieve optimization is the generation of integers instead of doubles by wrapping for instance the variable \( M \) with the construct \( CT\_intfix(M) \). One result is for instance that integer division can be realized within the M-code [9].
Apart from M-code annotations, compiler flags, have been introduced to MCS. The flag `-safe` leads to additional runtime checks added within the C-code, where `-fast` removes runtime checks. By leaving these flags away, the program contains checks such as for instance dynamic extension. One reason why these runtime checks are used is for instance to detect if a vector has been indexed beyond its runtime size. Another example is checks for runtime occurrence of negative numbers in functions like `sqrt` or `log`. In case the input argument to such a function is not already complex valued, an error message is displayed by default if a negative number occurs. This problem links back to the generation of complex numbers described at the end of section 1.2.2.

The readability of generated code is important and the programmer can therefore decide to have the original MATLAB source and/or comments in the C code. As a result, first comments and original source as C-comments are put into the generated code almost line-by-line directly followed by the corresponding C translation. In this way manual optimization is facilitated.

In contrast to the compilers described in section 1.3, MCS can be used to generate fixed-point C-code. As it originally comes from RMS, Fixed-point code generation is facilitated through fixed-point constructs added to the MATLAB prompt [11]; an example fixed-point M-code can be seen in appendix D.2.

The generation of a MEX version of the program to be translated can be invoked by the compiler flag `-mex`. In [10] an overview of the by MCS supported subset of MATLAB is given:

- Double, complex, logical and fixed-point types and according arithmetic
- Vector, matrix, arrays and structures
- Local, global and persistent variables
- Over 300 MATLAB functions (including signal processing, communications and image processing toolbox functions)

In reference to [10] these features of MATLAB are not supported:

- Arrays must be pre-allocated before loops
- Cell arrays, class references, function handles
• Recursion, Sparse matrices
• Functions such as `eval`, `feval`, `assignin`, and `evalin`
• Plotting functions and file I/O

4.2 Embedded MATLAB

Embedded MATLAB (EML) is a commercial product launched by the MATLAB designer MathWorks. This tool is comprised of two products: 1) Embedded MATLAB MEX (EMLMEX), which as RMS is developed to speed up MATLAB simulations; 2) EMLC to generate C-code. Together with the Simulink add-on called Real Time Workshop, the graphical engineering tool from MathWorks named Simulink can generate C-code. Real-Time Workshop is the origin of EML which leads to EMLC still being dependent on Simulink. The original idea was to enable C-code generation from Simulink designs, which has been extended to C-code generation from M-files.

Opposed to MCS, EMLC focuses on embedded C-code that does not require dynamic memory allocation and is optimized to be deployed on embedded targets. A drawback is due to the occurrence of MATLAB functions and maybe also user designed reference models requiring dynamic memory allocation, the MATLAB language is more restricted by EMLC. In order to generate static code, not just type and shape but also size information must be given to each input argument. Therefore the generation of an algorithm reference in C applicable for arbitrary data sizes in the way it is provided by MCS, is not possible. An annotation for the static example given in 4.1 for EMLC could look like `assert(isa(M,'int32') && isreal(M) && all(size(M) == [1 512]))`.

EMLC does not inline the MATLAB source code as comments line-by-line to the corresponding C-code, but by default the comments used in MATLAB are also found on the correct spot as comments in the C-code. In that way, it is also possible to track which M-code has been translated to which C-code. However, this approach requires comments to each line of M-code.

Similar to Catalytic Inc., MathWorks offers a MATLAB tool to simulate fixed-point designs, which has been facilitated through their earlier product named Fixed-Point...
Toolbox. The MATLAB classes (u)int32, (u)int16, (u)int8 may be used to force EMLC generating C-code which holds variables of the desired data type and for instance to realize integer division \[23\]. However, due to specific static semantics of MATLAB integer classes need to be understood first \[23\].

Since EMLC is aimed to generate embedded C-code, a convenient way has been developed to define different target architectures. Consequently, the standard C data-types \textit{char}, \textit{short}, \textit{int} and \textit{long} can be defined and stored as a target object to be used for C-code generation \[23\]. It is also possible to specify, whether a signed integer right shift is an arithmetic right shift or not, for the target hardware. In addition, it is possible to define, how integer divisions should be rounded and the significance of the first byte of a data-word is provided. By default EMLC provides a hardware target definition denoted as \textit{rtw} describing a generic host computer. In this thesis, all benchmarks for EMLC are compiled to the \textit{rtw} target. If no specific target has been given via a compiler flag, a C MEX function is generated for convenient C-code verification and/or speed up of MATLAB. In \[23\] an overview of the features supported by EMLC is given:

- N-dimensional arrays, matrix operations, subcripting and structures
- Complex numbers, numeric classes such as 8-, 16- and 32-bit integers and characters
- Double-precision, single-precision
- Fixed-point arithmetic
- 270 MATLAB operators and functions

The outlined unsupported features in \[23\] are:

- Cell arrays
- Command/function duality
- Dynamic variables, global variables
- Java and objects
- Matrix deletion and sparse matrices
Floating-Point Code Generation

Investigations on floating-point code generation are executed through application of method one and method two described in sections 2.2 and 2.3. The chapter discusses the compilers described in chapter 4, namely Matlab to C Synthesis (MCS) and Embedded MATLAB C (EMLC).

5.1 Results from different levels of compiler optimization

Initial investigations utilize basic algorithms known as Bubble Sort and Binary Search. The test data is composed of a vector $M$ which holds 10,000 integer elements and an integer scalar $key$. A test program named Sort-Search sorts the vector $M$ and returns the position of the value held by $key$ after sorting. Appendix B shows both the original MATLAB program and the manual translation to C. As discussed in section 1.2.2, MATLAB contains intrinsics and toolbox functions, which can cause translation problems. In order to concentrate first on elementary translation capabilities of the compilers in question, Sort-Search consists only of control structures and the fundamental built-in functions $size$, $length$ and $ceil$. In addition, no vector operations or dynamic extensions are used within that program. In tables 5.1 and 5.2 all benchmark results are listed. Further, by following the benchmark number given in the tables, the corresponding visualization can be found in figure 5.1. The interpretation of the results is given in sections 5.1.1 to 5.1.4.
Table 5.1: Execution times in seconds of the Sort-Search program. The left column shows the execution time of the original interpreted MATLAB file. In the center the execution time of the hand translated version is displayed, where the result on the right hand side has been achieved through the GCC compiler optimization flag -O3. The corresponding benchmark numbers can be used to compare the results in figure 5.1.

<table>
<thead>
<tr>
<th>MATLAB Interpreter</th>
<th>Hand Coded</th>
<th>Hand Coded -O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>Exec. Time</td>
<td>Benchmark</td>
</tr>
<tr>
<td>1</td>
<td>5.45</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.2: Execution times in seconds achieved by C-code generated with MCS and EMLC. The left hand side shows pseudo data type, shape and size annotations to the translation execution time of MCS and EMLC that are shown on the right hand side. In figure 5.1 the results to the corresponding benchmark number are displayed.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>MCS</th>
<th>EMLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>Exec. Time</td>
<td>Benchmark</td>
</tr>
<tr>
<td>realrow(M)</td>
<td>2</td>
<td>1.65</td>
</tr>
<tr>
<td>realscalar(key)</td>
<td>3</td>
<td>1.52</td>
</tr>
<tr>
<td>realrow(M)</td>
<td>4</td>
<td>1.42</td>
</tr>
<tr>
<td>realscalar(key)</td>
<td>-fast -retain_src_off</td>
<td></td>
</tr>
<tr>
<td>introw(M), size[1 1000]</td>
<td>7</td>
<td>1.37</td>
</tr>
<tr>
<td>intscalar(key)</td>
<td>8</td>
<td>1.27</td>
</tr>
<tr>
<td>introw(M), size[1 1000]</td>
<td>-fast -retain_src_off</td>
<td></td>
</tr>
<tr>
<td>As benchmark 8 &amp; 9 compiled with gcc -03</td>
<td>11</td>
<td>0.51</td>
</tr>
<tr>
<td>As benchmark 8 &amp; 9 compiled for MEX</td>
<td>13</td>
<td>0.63</td>
</tr>
</tbody>
</table>
5.1.1 Reference code

The translation of Sort-Search is first evaluated by applying method one described in section 2.2, which concentrates on C code generation for applications requiring algebraic reference code. This C model should be as generic as possible to enable simulation runs on different data type, shape and size without re-compiling. In contrast to EMLC MCS, as discussed in section 4.1, allows generation of dynamic code, which means that dynamic memory allocation is applied. This means that EMLC could not be used for the first investigation which considered whether C-code is, in theory, usable for any vector length. Annotations to the M-code required for generation of this generic C-program by MCS are: mbrealrow(M); mbrealscalar(key). Without the use of any flag, the compilation results in 135 lines of code containing the original M-code as comments and runtime checks for unsupported dynamic array extensions. If such an extension is hit during runtime, program execution stops and an error message pointing to the specific M-file and line number is displayed. As a result, this error can be found quickly in the M-code. This generic and run-safe attribute of the code is, as can be seen for benchmark 2 in table 5.2, paid with the price of the worst execution time. An approx. 0.1 seconds faster result (speed up = 1.08) is achieved by using the compiler flag -fast,
where the dynamic extension checks are removed for benchmark 3. With the flag `-retain_src_off` all M-code comments are removed, which results to 76 lines of code.

The next step of optimization again shows a approx. 0.1 seconds faster result (speed up = 1.08) for benchmark 4. Due to the added size information, C-code with comparable M-code annotations for EMLC could be generated this time. Such annotation for EMLC could look like:

```matlab
function middle = sortSearch(M, key)
    assert(isa(M, 'double') && isreal(M) &&
           all(size(M) == [1 10000]));
    assert(isa(key, 'double') && isreal(key) &&
           isscalar(key)); ...
```

and for MCS:

```matlab
function middle = sortSearch(M, key)
    mbrealrow(M); mbsize([1 10000], M);
    mbrealscalar(key); ...
```

At first glance the performance of EMLC for benchmark 5 in table 5.2 could be seen as negative compared to the translation result of MCS. However, investigation of the code generated by EMLC and MCS shows that in the MCS case a second array of the same length of vector `M` is created during runtime. Thus approx. 20,000 instead of approx. 10,000 `doubles` are held in the memory. Such a difference in memory usage does not matter for an algebraic reference application, but can have a significant effect on the implementation on embedded target hardware. In this respect, EMLC’s design to generate embedded code is “shining through”. With 89 lines the code length of the EMLC result is, without comments from the original M-code, slightly longer than the production by MCS which has 76 lines.

5.1.2 Target code

The vector `M` and the scalar `key` hold `integer` numbers, where the vector `M` has a length of 10,000 elements. To apply method two, described in section 2.3 it is required that the input arguments of the relevant MATLAB function are declared as accurately
as possible. The resulting pseudo annotation can be seen in table 5.2 for benchmarks 7, 8 and 9. MCS continued producing static memory allocation for a second array of again 10,000 elements. Therefore, benchmarks for Sort-Search EMLC required approximately half of the memory compared to MCS. EMLC does not come with flags to include or exclude runtime checks. From the start, each C-file generated by EMLC does not contain runtime checks such as for MCS. Since among other things MCS can provide C-code to be run on different array sizes, such runtime checks are required. The comparison of benchmark 8 (MCS -fast -retain_src_off) to benchmark 9 (EMLC) where MCS is 0.15 seconds faster, shows up the memory usage to execution time trade-off, discussed in section 2.3. Appendix B.3 illustrates the corresponding MCS translation and appendix B.4 shows the EMLC version.

5.1.3 MEX and GCC -O3

As described in section 1.2.2 MEX is used to interface for instance C code to MATLAB functions. This way of interfacing can be used to speed up MATLAB simulations. RMS and EMLMEX use MEX to speed up slow MATLAB functions through compilation to C. A known execution issue for the MATLAB interpreter are loops, where significant speed up can be achieved through compilation. MCS seems to use the same compiler technology as RMS does, with the difference that MCS actually provides the generated C-code to the user. EMLC and EMLMEX seem to correlate, in terms of compiler technology, in a similar way as MCS and RMS correlate. Due to containment of many loops, Sort-Search is significantly speed up by EMLC and MCS, where MCS accelerated the program even more (benchmark 13) than it was accelerated for benchmark 9. This difference of benchmark 9 to benchmark 13 leads to the suspicion that MCS uses some C compiler optimization flag like GCC’s -O3. The reason for EMLC performing dramatically slower could be, MathWorks’ run-time safety checks for EMLC generated MEX files.
5.1.4 Conclusions to initial tests

Due to the fact that Sort-Search is mainly comprised of loops, both compilers, EMLC and MCS, generate a significant speed up compared to the MATLAB interpreter. The number of code lines generated by the translation tools and a hand coded version do not differ noteworthy. EMLC can not generate code to be used for different vector length, but assures to not allocate memory dynamically. This corresponds to the fact that dynamic memory extension checks of such kind are not to be integrated in the C-code. MCS uses approximately the double amount of memory compared to EMLC, but achieves a better speed up compared to the MATLAB interpreter. The MEX function generated by MCS is considerably faster than the comparable EMLC version. In this particular case, MCS produces a MEX function close to the speed up achieved through compilation by GCC using compiler flag -O3. Consequently, for both, MEX function or reference C-code, MCS would be the best choice for simulation and verification runs on a PC: instead, for the implementation on target hardware, EMLC is the best choice. The hand coded version uses approximately the same amount of memory as EMLC, but the performance of the hand coded version is around 0.2 seconds faster (speed up = 1.45). Still, both compilers have proven to manage the compilation of basic control structures without problem and with an interesting performance. Especially for the simulation and verification speed-up and the generation of intermediate C-code shows a promising potential.

5.2 Benchmarks on translation of frequently used MATLAB intrinsics and toolbox functions to target code

The investigation on some MATLAB intrinsics and signal processing functions that were supported by both EMLC and MCS, should demonstrate how well the compilers in question can deal with different data types and rather basic functions. Reference code generation is not regarded for this study on MATLAB intrinsics. The author finds
it of interest to study how much automatic, as well as manual, translation to C can speed up execution compared to an original MATLAB execution. In order to study the performance difference between automatic translation and manual translation method two, described in section 2.3 is followed. Since EMLC by default does not contain any runtime checks in case of the compilation to the rtw target, the compiler option -fast is used to exclude these checks for MCS too. C-code compilation is executed with the GCC compiler optimization flag -O3. All speed-up results are illustrated in figure 5.2. The number of lines of code produced by each translation, can be viewed in table 5.3. None of the M-code contain any comments, which results in EMLC lacking comments from the M-code in generated code. MCS has been used with the flag -retain_src_off and therefore does not contain any original M-code in the generated C-files. As a result, the lines of code produced by both studied compilers are comparable. Sections 5.2.1 to 5.2.6 discuss setup of, and results from, the different benchmark algorithms.

5.2.1 The function: sort

In terms of functionality the sort algorithm used here does the same as Sort-Search described in section 5.1. The difference is that this time, the MATLAB built-in function
sort is utilized instead of an implemented Bubble Sort algorithm. Unfortunately the MATLAB function \textit{find} is not supported by EMLC, otherwise this function could have replaced the Binary Search applied in \textit{Sort-Search}. As test data an integer vector of length 1,000,000 and an integer scalar holding the search key are applied.

For both compilers, the results in figure 5.2 show a negative speed up compared to the MATLAB interpreter. This effect is probably due to the vector units in modern PC processors, which the MATLAB interpreter makes use of. MATLAB intrinsics can in general be considered to be optimized for PCs. The study of speeding up MATLAB simulations through compilation is not within the scope of the current thesis. Therefore the C-code is compiled without the \texttt{GCC} optimization for vector units. Embedded targets normally do not contain vector units, which is an additional reason to not apply vectorization optimization on C-code compilations. The visualization of speed-up to the MATLAB interpreter should only be of informative nature for the interested reader. In addition, the comparison of EML, MCS and manual translation performance is not influenced through using the interpreter’s performace as straightedge for speed-up computations. Also the faulty conclusion that interpretation is generally a faster way to execute programs than using compiled programs should not be drawn.
In this particular case EMLC achieves a better result than Catalytic Tools. Both compilers allocate space several times for the whole array of data for temporary computations. MCS is probably slowed down due to the dynamic memory allocation of 3 times the whole array of 1,000,000 elements, which can not be avoided with the optimization options available. A manual translation by using the Quick Sort algorithm achieves a positive speed up compared even to the MATLAB interpreter, where extra memory is neither statically nor dynamically allocated.

5.2.2 The function: fft

MATLAB's FFT function seems to be suitable for vectorization, which leads to a negative speed up of both C-translations. To verify the accuracy of the FFT computation, it is followed by an inverse FFT and the rounding noise ratio between input and output vector is printed out. MCS supports the compilation of the arbitrary length of the input vector to a FFT computation. Due to the requirement for EMLC to have an input vector with the size to be a power of two, a test vector holding 1,048,576 doubles has been chosen. This time Catalytic Tools shows better speed performance, although dynamic memory allocation is still applied. EMLC and MCS provided computation accuracy with an SNR of approximately 350 dB, which corresponds probably to the machine’s precision. The possible complex output of FFT operations requires extra attention, since ANSI C does not contain a complex data type. In section 5.2.4 the complex data representation in C is discussed in more detail.

5.2.3 The function: conv2

Among other things, two dimensional convolution is applied in the field of image analysis. Owing to the function’s operation on matrices, the row- to column-major issue discussed in section 1.2.2 has to be addressed for translation. In this respect MCS allocates space dynamically to transpose the data matrices and runs them through C-code following MATLAB’s column-major. EMLC just generates column major code by requiring the engineer to take measures accordingly. The use of the compiler flag -row_major_off for MCS disables automatic transpose code generation. Two 100 × 100
double precision matrices are used as test data. To render meaningful benchmark results, code from both compilers is generated in order to take input arguments and deliver output arguments in column-major fashion. The input data to the generated functions is transposed by the driver described in section 2.2 and shown in appendix A. In that way the computation result for both compilers could be verified to the MATLAB outcome. In terms of execution time, EMLC and MCS could achieve a slight speed up, where the difference between them is negligible.

5.2.4 The function: xcorr

Cross-correlation is a similarity measure between two signals. On this certain stage of benchmarking, translation to code working on complex data was of interest. MCS holds complex data by default on two arrays, where one array represents all real values and the other array contains the corresponding imaginary values. However, EMLC defines a structure comprising of a real and an imaginary value. Both ways of data representation have advantages and disadvantages for different computations [9]. The declarations of function input arguments for complex values exist for EMLC as well as for MCS and do not require additional effort to generate “complex” C-code. As test vector 10,000 normal distributed pseudo random complex values held in the MATLAB complex datatype are used. The computation of the auto-correlation seems to particularly favor the MATLAB interpreter, as illustrated in figure 5.2. This could be due to the fact that the MATLAB interpreter might executes the auto-correlation in the frequency domain and MCS as well as EMLC might translate to the time domain. In terms of speed EMLC performs slightly better than MCS.

5.2.5 The functions: inv, mldivide, transpose, filter

Matrix inversion, MATLAB left divide (solving an equation system), matrix transpose and filter, are translated by EMLC and MCS without issues. Functional correctness is verified to the MATLAB interpreter result. As for all functions described within section 5.2 MCS and EMLC take turns generating the faster code. Also in terms of code lines produced, there was no difference worth to mention between the two compilers. A
plus for EMLC and further for target implementation is the assurance of static code generation, where automatic transpose of data in matrix form by MCS was favored in case of reference applications.

5.2.6 The functions: mtimes, plus

One of the most frequent operations in a MATLAB program are probably the operations plus and multiply. These operators are overloaded in MATLAB, for instance, real scalars as well as for complex matrices. As visualized in figure 5.2, the application of two $200 \times 200$ (mtimes1) complex matrices could be slightly speed-up by translation to C. Matrices of size $400 \times 400$ (mtimes2) and bigger, however, show a similar pattern of negative speed up. This dependency on data size could be due to a caching issue or that the MATLAB interpreter uses JIT compilation to speed up execution on bigger data. However, the main focus for this thesis is the difference between EMLC, MCS and the hand-coded version. In this respect, no “real” difference between EMLC, MCS and the hand-code can be seen through varying data size. Also the plus operation on complex matrices of size $600 \times 600$ (mtimes3) performed favored of the MATLAB interpreter.

5.2.7 Discussion of results from MATLAB intrinsics and toolbox functions

Combining all results, none of the two compilers in question can be favored. Sometimes MCS delivers better results and sometimes EMLC. The same applies to the lines of C-code generated, where a manual translation delivered clearly better results. However, automatically generated code by EMLC as well as MCS is readable. Viewed together with the result described in section 5.1.4 it can be concluded that both compilers seem to support the “basis” of MATLAB in an efficient manner: a translation by hand can not clearly deliver better results. The code annotations are simple and no translation issue is faced on functions supported. However, some documented restrictions can appear, such as for example FFT in the EMLC case in section 5.2.2. A plus for EMLC is the guarantee of static memory allocation for target implementations.
5.3 Compilation tests on IEEE 802.16d (WiMAX) functions

The aim of the study described in this section is to demonstrate the effort required to successfully compile to reference and to target code. Investigation on Worldwide interoperability for Microwave Access (WiMAX) code refers to the benchmark code description in section 3.2. The translation of a reference model from M-code to C-code can be viewed from several aspects. A first study of the three WiMAX functions Interleaver, CC-Encoder and a partial transmitter Burst follows method one described in 2.2. The Burst refers to a top level function invoking the Interleaver and the Encoder as part of a transmitter chain. Consequently, the interoperability of two automatically translated functions can be investigated on. Method two, as utilized for the studies in sections 5.1 and 5.2 proves to be a reasonable method to investigate on automatic target code compilation. WiMAX code is also studied in terms of translation to embedded code, where method two is applied. Since EMLC does not support dynamic memory allocation, the generation of reference code to be interfaceable with data of arbitrary size is not possible. As a result, only benchmarks for MCS reference models could be executed. Additionally, the matter of assessment is speed-up achieved through the MEX interface. All results are illustrated in figure 5.3 and for more detail in table 5.4. The generated lines of code can be seen in table 5.5.

5.3.1 Interleaving

Interleaving is used for digital signal transmissions to prevent burst errors. Basically, data bits are scrambled in a particular pattern before transmission. At receiver side these data bits are then unscrambled by following the pattern applied at the transmitter side. In the event of an errornous transmission, it is often the case that only parts of symbols are affected and can then easily be reconstructed [17]. As test data a bit-block matrix $\mathbf{x}$ of size $192 \times 10000$ holding either 1 or -1 is generated. These data resembles a transmitter burst in BPSK modulation mode, as specified in [1]. Two other input arguments required by the interleaver function are an integer to determine
the modulation scheme $\text{Ncpc}$ (BPSK = 1, QPSK = 2, 16-QAM = 4, 64-QAM = 6) and a character-string to determine the direction of dataflow $\text{dir}$ (’tx’, ’rx’).

MCS translates the WiMAX Interleaver, by annotating $\mathbf{x}$ of type integer and shape matrix, $\text{Ncpc}$ as integer scalar and $\text{dir}$ as character row, straight away. The generated code can be used as C reference model on any modulation scheme, data shape and size. Verification with the data generated by the MATLAB interpreter was successful. Speed up performance and the generated lines of code as given in tables 5.4 and 5.5 can be considered as reasonable, in those cases where the GCC flag -O3 came not into use.

The next step was to find out how much the Interleaver compilation can be optimized by EMLC and MCS. Therefore, the size information for $\mathbf{x}$ of 192 × 10000 has been annotated to the MCS annotation. Again, translation could be executed without a problem. As the same annotations for EMLC have been made, issues due to the special static semantics of integer MATLAB classes have been faced [23]. The annotation in question was:

\[
\text{assert} \left( \text{isa} (x, \text{'int32'}) \&\& \text{isreal} (x) \&\& \text{all} (\text{size} (x) == [192 10000]) \right);
\]

By changing back to the data type double, the issue could be resolved, but another issue surfaced. In order to generate static code EMLC needs to have all variables able to change the size of a data array into a constant. The relevant variable was $\text{Ncpc}$, which determines the modulation scheme. Owing to the need of $\text{Ncpc}$ to be a constant, the translated function to C code by EMLC could just support one modulation scheme. To generate code for all modulation schemes 4 different versions of the Interleaver function would have to be generated and dealt with. The third issue with EMLC was faced at a switch-case statement composed as followed:

\[
\text{switch} (\text{dir}) , \\
\text{case} \ '\text{tx}' , \ y(jk+1,:) = xz ; \\
\text{case} \ '\text{rx}' , \ y = xz(jk+1,:) ; \\
\text{otherwise} \ \text{error} ( '\text{dir must have values }' \text{‘tx’ or ‘rx’} ) ; \\
\text{end}
\]

EMLC requires the evaluation of switch-case statements as well as if-else statements on integers. This implies to rewrite the particular statement and to redefine the $\text{dir}$
input argument as integer. Having the WiMAX chain in mind, this change would have to be propagated throughout the whole chain, in order to invoke a first translation. Before translation can successfully be started, the MATLAB function error has to be outcommented, since it is not supported by EMLC. After all these changes EMLC finally compiles the interleaver to static C code. A big drawback is that the generated code contains a two-dimensional array of $192 \times 10000$ double precision elements, where just 1 and -1 are to be held by the matrix representation. It requires wrapping of computations within the M-code by following the static semantics of MATLAB integer classes to finally generate the data matrix $x$ as C-type int.

Investigation on the MCS C-code generation shows that int is generated as input data-type. The output argument, however, is a matrix pointer of type double. Consequently, the GUI of Catalytic Tools showed at which point the data matrix started to be inferred as type double. Wrapping the parts shown up in the GUI with the construct CT_intfix leads to the propagation of the integer data type through the whole code, which is verified in the C-code. Finally, MCS and EMLC achieve meaningfully comparable translations, where MCS uses the -fast and -retain_src_off flags. C-code compilation is executed with GCC flag -O3. The corresponding results are shown in tables 5.4 and 5.5. The MCS compilation still contains dynamic memory allocation, even if $\text{Ncpc}$ is declared as a constant and dynamic code is not required. To conclude, translating the Interleaver can be a question of minutes with MCS and a question of hours with EMLC. On the other hand, EMLC guarantees to generate static code and showed a better execution time. The production of lines of code are significantly lower by MCS. Since EMLC also supports the translation of the MATLAB classes int16 and int8, the translation with int16 gains on speed up and halves the required memory space. Compilation with int8, however, leads to erroneous data without any error message during runtime or compile time. This again proves that MATLAB integer classes are to be used with care in this respect. The author considers that support for 8 and 16 bit datatypes with reasonable error checks and ease of use for both, MCS and EMLC could be of advantage. The declaration of $y = \text{nan}(\text{size}(x))$ is no problem for reference code generation, but for compiler optimization for MCS. The change of the MATLAB nan function to the zeros function, enables better type inference for MCS, as EMLC
does not have any issues with \textit{nan}.

5.3.2 Convolutional encoding

In addition to interleaving convolutional encoding is also a measure against erroneous transmission. This form of encoding increases the message length by adding redundancy \cite{17}. The reference code generation of the \textit{CC-encoder} functioning with MCS, hits the problem that the MATLAB functions \textit{poly2trellis} and \textit{convenc} are not supported by MCS. Through Catalytic support, a workaround can be applied and reference code generated without any other measures to be taken. According to Catalytic, the problem of translating these functions is their use of cell arrays. Also EMLC can not translate these functions, but the workaround supplied by Catalytic can be also utilized by EMLC. As for the \textit{Interleaver}, the modulation scheme has to be statically determined for compilation with EMLC. Also similar issues as with MATLAB \textit{integer} classes are faced and a \textit{switch-case} statement need to be rewritten, as it was with the \textit{Interleaver} function. In Additon, the MATLAB function \textit{dec2bin} is not supported by EMLC, which is basically due to the dynamic data dependency of the functions output argument. With the help of MathWorks support, a static \textit{dec2bin} workaround for the 8-bit output can be programmed. Finally, also EMLC could translate the \textit{CC-encoder}. By applying \textit{int32} classes for EMLC and \textit{CT_intfix} constructs for MCS, a meaningful comparison of optimized C-code in alignment to the input argument annotations given could be generated. This time, the version produced by MCS achieves greater speed up, which could be due to the manually rewritten version of the \textit{dec2bin} function for EMLC. As for the \textit{Interleaver}, even MCS' \textit{GUI} showed successful size inference throughout the code by setting the modulation scheme static, dynamic memory allocation for intermediate computations could not be avoided. The lines of code created by EMLC are approximately double as much as produced by MCS.

The attempt to declare \texttt{Ncpc} as \texttt{character} can be compiled by MCS. To declare \texttt{Ncpc} as \texttt{character} should be possible, since the values \texttt{Ncpc} can have are between 1 and 6. The output values, however, are erroneous without any error message during compilation execution. After that, the code generated with the MCS flag \texttt{-safe} points out
Table 5.4: *Speed-up to the original MATLAB interpretation of WiMAX interleaver, encoder and partial transmitter burst translation*. A visualization can be found in figure 5.3. MEX stands for the corresponding compiler option to generate and compile C-code to be interfaced with the MATLAB prompt.

<table>
<thead>
<tr>
<th>Function</th>
<th>MCS Ref</th>
<th>MCS -fast</th>
<th>MCS MEX</th>
<th>EMLC</th>
<th>EMLC MEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaver</td>
<td>1.06</td>
<td>13.74</td>
<td>1.97</td>
<td>17.23</td>
<td>2.00</td>
</tr>
<tr>
<td>CC-Encoder</td>
<td>2.00</td>
<td>4.46</td>
<td>3.07</td>
<td>0.85</td>
<td>0.75</td>
</tr>
<tr>
<td>Burst</td>
<td>1.32</td>
<td>3.06</td>
<td>2.44</td>
<td>0.98</td>
<td>0.67</td>
</tr>
</tbody>
</table>

Table 5.5: *Generated lines of code for selected WiMAX function compilation by given translation option.*

<table>
<thead>
<tr>
<th>Function</th>
<th>M-File</th>
<th>MCS Ref</th>
<th>MCS -fast</th>
<th>EML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaver</td>
<td>57</td>
<td>515</td>
<td>90</td>
<td>368</td>
</tr>
<tr>
<td>CC-Encoder</td>
<td>64</td>
<td>632</td>
<td>237</td>
<td>434</td>
</tr>
<tr>
<td>Burst</td>
<td>150</td>
<td>1219</td>
<td>580</td>
<td>875</td>
</tr>
</tbody>
</table>

the error at runtime (division by 0 at M-code line 43). In this case the MCS flag -safe has been proven useful. According to [9], each code should be inspected for run safety with this flag.

5.3.3 Transmitter burst on interleaver and convolutional encoder

The interoperability of *Interleaver* and *CC-encoder* function is studied through a top level function called *Burst*. In addition the compilers support for *structures* is tested by feeding *Interleaver* and *CC-encoder* by means of a “multi-level” *structure*, e.g. in MATLAB defined as `y.data.rs = x`. In this respect Catalytic’s [GUI] proved to be helpful to follow type, shape and size inference through the different levels of the *structure*. In contrast to MCS, EMLC does not allow the addition of *structure-fields* once the *structure* has been used the first time. As a result, in addition to the for EMLC compilation prepared *Interleaver* and *CC-Encoder* function, all *structure-fields* in the
5. Floating-Point Code Generation

Figure 5.3: WiMAX function speed-up through translation for different compilation options on a base 10 logarithmic scale. Detailed speed-up values can be found in table 5.4. MEX stands for the corresponding compiler option to generate and compile C-code to be interfaced with the MATLAB prompt.

M-code of the Burst function, had to be declared at the top for EMLC. Compilation with EMLC takes approximately 90 seconds, which is considerably more time than the approximately 3 seconds needed by MCS.¹

5.4 Implications of IEEE 802.16d transmitter reference code generation

It is of interest how close the compilers in question are to a so-called “MATLAB to C one click translation”. The WiMAX code described in section 3.2 is considered suitable, due to its resembling the practice. Each compiler, both EMLC and MCS, is tested on all functions of the transmitter chain. Therefore only the least necessary amount of M-code annotations are made and arising error messages logged.

¹Both translations have been executed on the platform described in chapter 2
5.4.1 Catalytic Tools

randomize.m: This function can not be compiled due to the use of the empty matrix ‘[ ]’. In addition, dynamic array extension came into use within a variable dependent loop, which can not be translated by MCS. As a possible solution, the slice-hoisting described in section 1.3.8 could be of interest for implementation to MCS.

rs_encode.m: The RS-Encoder contained with gf, rsenc and rsgenploy three unsupported functions.

cc_encode.m: Two unsupported functions poly2trellis and convenc are the problem for translation.

interleave.m: Compiles successfully, test run and verification through MEX succeeds too.

modulate.m: Compiles successfully, test run and verification through MEX succeeds too.

scmap.m: In this function again a dynamic array extension is the problem for translation.

amble.m: As for scmap a dynamic array extension can not be compiled.

pilot.m: The MATLAB function mpower is not overloaded for matrices in the MCS function-set.

5.4.2 Embedded MATLAB

randomize.m: As for MCS the empty matrix ‘[ ]’ can not be dealt with. In addition, the operations & and | have to be replaced with && and ||. Also MATLAB function exist is not in the EMLC set. After declaring eml.extrinsic (’exist ’) [23], a problem comes up with an if statement evaluated on a character array.

rs_encode.m: The three MATLAB functions gf, rsgenploy and rsenc are not supported.
**cc_encode.m:** A `switch` statement is evaluated to a character array, leading to an error.

**interleave.m:** A variable needs to be declared as constant, due to an array whose size depends on that variable.

**modulate.m:** Function `int2str` is not supported by EMLC and modulation scheme must be static.

**scmap.m:** Same error from the start as with the Inerleaver, variable needs to be declared as constant.

**amble.m:** To define imaginary constants in EMLC $3*\imath$ must be changed to $3\imath$.

**pilot.m:** MATLAB functions `ismember` and `find` are not supported.

### 5.5 Conclusions on WiMAX translation performance

The benchmarking of the WiMAX transmitter chain provided by Ericsson AB points out several problems with automated MATLAB to C-code translation. The most tedious problem is the missing support for required functions such as `poly2trellis`, `rsenc`, even for reference code generation with MCS. Due to this problem, it is often required to spend considerable amount of time in finding workarounds for functions. The goal of the benchmark described in section 5.3 is to demonstrate the effort to be spent to get reference or target code generated. On the other hand, section 5.4 illustrates an approach to investigate on the ease of translation. The main conclusions drawn for MCS and EMLC are summarized in table 5.6.

#### 5.5.1 The compiler MCS

Catalytic Inc.’s tool MCS shows potential for reference code generation. Out of eight functions, two can be compiled straight away. Three other functions can not be translated, due to dynamic array extension. The remaining three functions contain unsupported MATLAB intrinsics. Next to the functions mentioned in section 5.4.1 and
dynamic array extension, no further complications for reference code generation with
MCS is faced in the present thesis. However, for instance through annotation of the
size of the variable \( y \) in the WiMAX function \( scmap.m \), also dynamic array extension
problems can be fast solved. So remaining as problematic for MCS are these three
functions, which make use of unsupported MATLAB built-in functions.

The solution of annotating the type and the shape of an input argument but not
the size, proofs to be efficient. Often different data length is fed to an algorithm for
simulation and verification runs, but the shape of test data hardly changes, e.g. from
a vector to a matrix. Normally two or three short statements per top-level function of
an algorithm are required to achieve a successful compilation.

Target code generation can not be recommended with MCS. The discussion of fixed-
point constructs aside, the smallest \( \text{C} \) integer type possible to generate is \( \text{int} \). Moreover,
dynamic memory allocation can not be avoided and memory is generally not in focus
in terms of minimization. As shown in figure 5.3, a considerable speed up is achieved
in case of the interleaver, but the corresponding C-code still contains statements for
dynamic memory allocation. Further, a matrix of size \( 192 \times 10000 \) holds only the values
1 or -1. Still each element is of type \( \text{int} \).

5.5.2 The compiler EMLC

The strength of EMLC is clearly the generation of target code. Not one function of the
WiMAX chain in question could be translated. The EML language, as given in [23],
requires too many changes in the M-code of a complexity such as a WiMAX chain to
quickly achieve a compilable M-code version. In contrast to MCS, common functions
like \( \text{dec2bin}, \text{disp}, \text{int2str} \) are not supported. Also, for simulation and verification runs,
the generated code can be applied only to one fixed modulation scheme and to one
defined input data size.

In the interleaver case, as shown in figure 5.3, a considerable speed up could be achieved
with EMLC. Further, the code suits embedded target deployment. However, the auto-
mated embedded C-code production implies considerable restrictions on the MATLAB
language, as demonstrated in section 5.3. Taking the IEEE 802.16d standard as example, the advantage of writing programs in EML syntax is: target code can be produced directly from the MATLAB prompt. Consequently, only one source code needs to be maintained. This can speed up product design iterations considerably. A drawback is that, while writing the algorithm, an engineer also needs to have target specifications in mind. This can influence the designer’s creativity and further the quality of algorithms designed.

Also, a later translation of the M-code to EML is not favorable in the WiMAX case. Next to the more specific functions as highlighted in section 5.4 and dynamic array extension to be addressed, the generation of 4 different versions of the code to support each modulation scheme is required. Further the code is to be rewritten in order to follow the static semantic rules of MATLAB integer classes. Work arounds have to be programmed for rather basic functions, such as `dec2bin`. Functions in use to print out information during simulation runs to the MATLAB prompt, like `warning`, `error`, `disp` have to be commented out. Finally, statements such as `switch-case`, `if-else`, evaluating on non-integers, have to be rewritten to evaluate on integers. This change can lead to the need, to refactor a considerable amount of code. From this perspective, the generation of an intermediate C reference for a complexity like WiMAX seems to be a better solution.
Table 5.6: MCS and EMLC comparative summary of performance on the WiMAX transmitter chain from Ericsson AB.

<table>
<thead>
<tr>
<th></th>
<th>MCS</th>
<th>EMLC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reference code generation shows promising potential.</td>
<td>Interesting for target code generation from the MATLAB prompt.</td>
</tr>
<tr>
<td></td>
<td>Common functions supported, e.g. also those which require dynamic memory allocation, or print to command line.</td>
<td>Problems with print functions and functions requiring dynamic memory allocation.</td>
</tr>
<tr>
<td></td>
<td>Little effort required to compile to a first version successfully (use of more common functions, no dynamic array extension).</td>
<td>Often many manipulations of the M-code required to initiate a successful first compilation.</td>
</tr>
<tr>
<td></td>
<td>Type inference graphically visualized, before compilation.</td>
<td>Type inference can just be seen in the C-code after compilation.</td>
</tr>
<tr>
<td></td>
<td>Forcing compilation to integers is straightforward.</td>
<td>MATLAB integer classes used follow special static semantics, which can be tedious.</td>
</tr>
<tr>
<td></td>
<td>Generated code readable: MATLAB source as comments in-lined, code follows a comprehensive notation.</td>
<td>Generated code suitable for manual optimizations (follows notation, MATLAB comments in-lined).</td>
</tr>
<tr>
<td></td>
<td>Often dynamic memory allocation used for intermediate computations, even all variable sizes could be inferred throughout the M-code.</td>
<td>Assures static memory allocation.</td>
</tr>
<tr>
<td></td>
<td>Direct hardware deployment needs more hand optimizations on the C-code.</td>
<td>The code seems to be more suitable for direct hardware deployment.</td>
</tr>
<tr>
<td></td>
<td>Only one size for MCS’ integer wrapping function available.</td>
<td>MATLAB integer classes supported: code for 8, 16 and 32 bit integers can be generated.</td>
</tr>
<tr>
<td></td>
<td>Good results on execution performance, although more memory is used.</td>
<td>Execution time and memory usage well balanced (fully supported code is fast).</td>
</tr>
<tr>
<td></td>
<td>Many options for optimizations, however, it needs experience to know how to apply these.</td>
<td>Compiler forces optimizations before compilation succeeds: Optimization potential on the M-code is shown up through error messages.</td>
</tr>
<tr>
<td></td>
<td>Short compile time.</td>
<td>Compile time can be considerably long.</td>
</tr>
</tbody>
</table>
Fixed-Point Implementation of a Digital Filter

Power consumption, silicon size and heat dissipation must be as low as possible for embedded processors; in addition, it is required to meet specified computation accuracy and performance. Floating-point data types are a convenient programming tool for all computations dealing with the fractions of integers. Figure 6.1 shows as example the partitioning of the IEEE float data type. The exponent determines the position of the decimal point, where the mantissa holds the corresponding fraction information of a decimal number. As a result, the exponent determines the range of a floating-point type, where the size of the mantissa determines the precision. After each floating-point computation, the exponent is automatically updated, and at the same time holds the best precision possible for a particular amount of bits. In that way the program designer does not have to think about mantissa and exponent handling, which in many cases simplifies programming tremendously. Floating-point data types can be applied as conveniently as any integer calculation. Where as floating-point computations are mostly no problem for a PC anymore, the additional computation power and/or silicon

![IEEE standard single-precision floating-point format.](image)

Figure 6.1: IEEE standard single-precision floating-point format.
size needed for floating-point operations and the resulting increase of heat dissipation is a problem in embedded systems especially. Thus it is too costly for many applications to have a Floating Point Unit (FPU) [18, chap. 6] in a digital signal processor. This means that the engineer must use fixed-point representation for decimal calculations.

Digital Filters are commonly found in the field of digital signal processing. The computer language MATLAB has become a popular inter alia in the signal and image processing domain [5]. Digital filters can be described as system function [28, chap. 7] given by

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_n z^{n-1}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_m z^{m-1}}$$  (6.1)

with B as the numerator filter polynomial and A as the denominator filter polynomial. In MATLAB, a floating-point filter can be realized by giving the coefficients of a filter system function to the built-in function \texttt{filter}. In section 5.2.5 the translation to floating-point code is dealt with. Both, MCS and EML, support compilation of the MATLAB \texttt{filter} function in floating-point. An attempt to quantize the \texttt{filter} function by the utilization of fixed-point constructs of the translation tools, does not work. A fixed-point implementation of a digital filter implies considerably more attention, than it does for floating-point.

### 6.1 Manual fixed-point filter design

The manual example design in the present thesis follows the method described in [20]. The target to be designed for is, as described for method three in section 2.4, a 16 bit processor with 32 bit accumulators. The goal of this example is to reveal potential difficulties implied by a fixed-point filter implementation. As figure 6.2 shows, the starting point is a filter specification normally not written by the software engineer. A floating-point filter model is then designed and often implemented in MATLAB. This floating-point model can then be implemented in a C reference that is, for example, part of a signal processing reference chain. This reference code can then be useful for the verification of a fixed-point implementation, or even to test a hardware design. Therefore, the automatic generation of reference code of MATLAB to C compilers
To continue with the manual filter design, the floating-point reference is taken to accomplish the fixed-point design. Design choices need to be documented and in case a design does not follow the specification, the process iterates back to the start as shown in figure 6.2. Finally a fixed-point reference design is implemented.

### 6.1.1 Design specification

**Bandpass filter**

- **Filter type:** Chebyshev Type 1
- **Filter Gain:** 1
- **Order:** 8
- **Passband:** 300 to 3400 Hertz (Hz)
- **Passband Ripple:** 0.5 dB
- **Sampling frequency:** 8000 Hz

**Digital signal processor:**

- **Architecture:** 16 bit
- **Accumulator:** 32 bit

---

1Information about filter specification can be found in [28].
6.1.2 Partitioning and coefficient computation

In order to compute the floating-point coefficients of the filter polynomials \(B\) and \(A\) as illustrated in equation (6.1), MATLAB’s built-in function \textit{cheby1} is used. The floating point reference code is the MATLAB’s \textit{filter} built-in function fed with the computed coefficients. This MATLAB intrinsic \textit{filter} function is goal to be translated to fixed-point C-code. There are Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, where the latter can be realized as the specified Chebyshev Type 1 [28, chap. 8].

Filter partitioning benefits floating-point designs, but can be considered to be crucial for fixed-point designs. The specification given in section 6.1.1 describes an eighth order bandpass IIR filter. The software engineer chooses how to partition the filter. Two possible ways to realize this filter are to create eight First Order Section (FOS)s or four Second Order Section (SOS)s. However, FOS have complex coefficients, which can be avoided with SOS for poles or zeros not on the real axis. These sections can then be interconnected in cascade, in parallel or in combination. Several approaches support the realization of SOS [31]. The implementation of the current example specification is carried out as four direct-form-II SOS. In figure 6.3 a direct-form-II SOS is illustrated.

6.1.2.1 Cascade filtering

One advantage of cascade filtering is that it saves on computation power. However, the filter propagates quantization errors from section to section. First, the SOS filter coefficients have to be calculated. For a cascade filter, these coefficients are found by computing the zeros and poles of the relevant filter transfer function. The zeros are the roots of the numerator polynomial \(B\) and the poles are the roots of the denominator polynomial \(A\). As the example filter is of order eight, eight zeros and/or eight poles are computed. In order to produce SOSs of flat magnitude response, two zeroes and two poles of minimal distance need to be grouped together. This approach is visualized in figure 6.4 sections two and four show a flat magnitude response; however, sections one and three contain a peak up to 20 dB of gain, which at the end leads to a loss of signal precision.
6.1.2.2 Parallel filtering

Parallel filtering has the advantage that it does not propagate quantization noise, but is more computation intensive. To gather the SOS coefficients for a parallel partitioned filter, partial fraction calculation is applied. The current example filter transfer function provides eight fractions. Fractions are computed together two by two, formulating second order sections. Consequently, four second order fractions which hold the SOS coefficients are produced.

6.1.3 Scaling

The limited range of fixed-point numbers requires scaling of the input data corresponding to the certain SOS. The current example filter design follows the approach presented in [19] to compute appropriate norms for scaling. There are three commonly used scaling norms $L_1$, $L_2$ and $L_\infty$, which can be ordered as $L_2 < L_\infty < L_1$. According to [19], these norms can be characterized as follows:

- $L_1$: Is considered as conservative scaling; prevents overflow, but often limits the dynamic range too much.
- $L_2$: The energy of a signal is considered; keeps the dynamic range best, but may lead to overflow.
- $L_\infty$: A worst case single sinusoid signal is considered; for signals with a wide bandwith, overflow occurs, but not very often.

6.1.4 Section ordering and coefficient scaling for the cascade filter

As depicted in figure 6.5 cascade filtering is a multiplication between the SOSs. To prevent error propagation best, it is important to order the sections ascending by scaling norm. SOSs have an external as well as internal filter gain, where the internal filter is regarded as the feedback system without the $B$ coefficients. In this respect the discussed norms need to be computed for each section without the $B$ polynomial
Figure 6.3: A direct form II second order filter section from [20], by permission from the author. The variable $q$ stands for internal filter memories to save intermediate computations.

too. Figure 6.3 illustrates, that the replacing of $B$ with the constant one leads to the internal filter system function. Then to avoid internal filter memory overflow, the bigger norm resulting from this process needs to be taken for input scaling. In addition to input scaling, the external filter section gain needs to be scaled back on the output data of the corresponding section. The external scaling norm is always used for this computation. This external scaling norms are denoted $\sim L$ in table 6.1. In figure 6.4, it can be seen that sections one and three are to be at the end of the filter chain. Indeed, the resulting norms in table 6.1 lead to the order of the sections as shown in figure 6.5.

There is no need to execute scaling computation on each data element after each filter section. The numerator filter coefficients as seen in figure 6.3 can be precomputed to scale back the external gain of the filter and to scale down the input data of the succeeding section. Only one actual scaling computation on the data is left to be executed at run time: before the input signal enters the first section.
Figure 6.4: Eights order IIR filter partitioning. The left column shows the magnitude response of a SOS, resulting from the in the right column illustrated zeros and poles.

Figure 6.5: Cascade SOS ordering with $B$ coefficient scaling. $s_x$ representing the internal and $s_{x_e}$ the external coefficient scalefactors.
Table 6.1: Filter norms for the cascade filter sections.

<table>
<thead>
<tr>
<th>SOS #</th>
<th>$L_1$</th>
<th>$L_\infty$</th>
<th>$L_2$</th>
<th>$\tilde{L}_1$</th>
<th>$\tilde{L}_\infty$</th>
<th>$\tilde{L}_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79.7582</td>
<td>19.8151</td>
<td>11.9125</td>
<td>4.0374</td>
<td>2.3313</td>
<td>0.9079</td>
</tr>
<tr>
<td>2</td>
<td>9.2628</td>
<td>7.9782</td>
<td>3.2127</td>
<td>2.0188</td>
<td>0.9908</td>
<td>0.9332</td>
</tr>
<tr>
<td>3</td>
<td>21.5527</td>
<td>5.1804</td>
<td>4.5250</td>
<td>4.2326</td>
<td>2.4082</td>
<td>1.0613</td>
</tr>
<tr>
<td>4</td>
<td>3.0570</td>
<td>2.5294</td>
<td>1.5367</td>
<td>2.1836</td>
<td>1.2925</td>
<td>1.1373</td>
</tr>
</tbody>
</table>

Figure 6.6: Parallel filter SOS interconnection; input vector and coefficient scaling with $s_x$ as internal and $\tilde{s}_x$ as external scale factors.

6.1.5 The parallel filter coefficient scaling

A parallel implementation of a filter is achieved by adding the output signals of the SOSs, as illustrated in figure 6.6. The advantage of parallel section interconnection is that errors from one section do not add to the error of another section. As for the cascade filter, the external gain of each SOS can be computed to the corresponding $B$ polynomials. In this respect extra computations have to be carried out on the input data in runtime. In a parallel filter, each section can be considered as a first section, which leads to the need for scaling computation on input data in runtime for each section. In table 6.2 it can be seen that not the internal but the external scaling norms are different, compared to the cascade filter. This leads to the change of only the $B$ polynomial but not the $A$ polynomial for the parallel filter.
Table 6.2: Filter norms for the parallel filter sections.

<table>
<thead>
<tr>
<th>SOS #</th>
<th>$L_1$</th>
<th>$L_\infty$</th>
<th>$L_2$</th>
<th>$\tilde{L}_1$</th>
<th>$\tilde{L}_\infty$</th>
<th>$\tilde{L}_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79.7582</td>
<td>19.8151</td>
<td>11.9125</td>
<td>1.3838</td>
<td>1.0217</td>
<td>0.2113</td>
</tr>
<tr>
<td>2</td>
<td>21.5527</td>
<td>5.1804</td>
<td>4.5250</td>
<td>1.3926</td>
<td>1.0212</td>
<td>0.2837</td>
</tr>
<tr>
<td>3</td>
<td>9.2628</td>
<td>7.9782</td>
<td>3.2127</td>
<td>1.6329</td>
<td>1.1198</td>
<td>0.6574</td>
</tr>
<tr>
<td>4</td>
<td>3.0570</td>
<td>2.5294</td>
<td>1.5367</td>
<td>1.4691</td>
<td>0.8562</td>
<td>0.8068</td>
</tr>
</tbody>
</table>

6.2 Results from a filter realization in fixed-point C-code

The fixed-point simulation of the parallel filter in MATLAB shows the best SNR for $L_1$ scaling. The cascade filter, in contrast has $L_\infty$ scaling as the best choice for the applied test vector. The test data is comprised of 1000 random fractional numbers between -1 and 1. An initial manual translation of the best scaling parallel and cascade filters from the MATLAB fixed-point reference design to C is completed fast. The parallel filter realized in C achieves a SNR of 54.49 dB where the cascade filter results in a SNR of 51.57 dB. These SNRs are computed by interfacing the C-code via MEX to the MATLAB prompt. The error print in figure 6.7 shows a slight periodic behavior of the filter. In this respect, several more optimizations can be applied. By dividing each input test vector element by four, the periodic behavior disappeared. This leads to the suspicion, that there is probably some noticeable overflow occurring in the filter. However, filter optimization is not in the scope of the present thesis. Due to the fact that the passband frequency of the example filters are used in phones, additionally tests with voice files on the filters are carried out. The resulting voice signals are clearly understood. The corresponding C cascade filter including MEX interface is shown in appendix C.

Bit accurate translation from MATLAB to C, however, can sometimes lead to complications. In order to demonstrate one of the challenges floating-to-fixed-point translation tools face, an example leading to a rounding problem is given. Depending on the target architecture, the realization of negative numbers can vary. Fixed-point negative
numbers can be denoted as A(a,b), where a = n - b - 1 for a n-bit binary number. A common representation is the signed two’s complement holding for a specific n-bit A(a,b) fixed-point binary number x, the value as given in equation (6.2)

\[ x = \left( \frac{1}{2^{b}} \right) \left[ -2^{n-1} x_{n-1} + \sum_{0}^{n-2} 2^{n} x_{n} \right] \]  

(6.2)

To simplify the example, the signed two’s complement of the 8-bit integer number -3 is considered, which is represented in memory as: 11111101. By applying the C right shift operator like -3 \( \gg \) 1, the resulting bit pattern is 11111110, which represents a signed two’s complement 8-bit integer -2. When the function bitshift(-3, -1) in MATLAB is applied instead, the result is -1. The same result as for MATLAB is achieved by computing \(-3 \times 2^{-1} = -1.5\) where integer truncation leads to -1. However, the actual meaning of a C shift operator depends on the embedded C compiler.

### 6.3 Fixed-point design aids from Catalytic Tools

The fixed-point constructs described in [11] to simulate fixed-point designs in MATLAB using RMS are also supported for translation to C with MCS. In addition, RMS comes with a tool to collect statistical information about data on a certain variable during simulation runs. This RMS add-on, denoted by Catalytic as Profiling Tool, displays statistical information such as histogram, minimum, maximum, mean and standard deviation about each variable after simulation. Information about the location of each variable in terms of function and M-code line number is also given. Running Catalytic’s Profiling Tool on the infinity norm cascade filter, discussed in section 6.1, shows a slight overflow on the internal memory variable w\(^{1}\), as can be seen in figure 6.8. Consequently, detailed information about a particular simulation run can be gathered. In contrast to the filter scaling norms described in section 6.1.3, this statistic tool seems to be strongly dependent on test data fed.

\(^{1}\)Illustrated in figure 6.3
Figure 6.7: Comparison of the example manual fixed-point cascade filter with the MATLAB floating point reference. The plot at the top shows that the first 50 milliseconds of both signals overlapped. The plot in the center shows both signals in the spectral domain. At the bottom, the error between the reference signal and the fixed-point signal is plotted per sample.
6.3.1 The squared average of a vector

To start up a test of Catalytic Tools on fixed-point conversion, the program described in section 3.3.1 is utilized. The Catalytic Tools fixed-point constructs seem to offer a complete set for fixed-point arithmetic. This includes for instance different rounding and overflow modes, arithmetic and decimal point shift, a mode for software as well as hardware targets and different warning modes [11]. Test data is composed of a vector increasing from -9.99 to 9.99 by 0.01 steps resulting in a length of 1999. The reference result of the example algorithm on these data is 33.3.

The implementation of the algorithm in single precision floating-point C by hand computes the value 33.2814. In contrast, a 32-bit fixed-point construct achieves 33.2999. This result links to the increase of precision, since space for an exponent as shown in figure 6.1 is not required. If the hardware specification for method three in section 2.4 is followed, the result 32.9124 is achieved, which could probably be accomplished through the 32-bit accumulator. Due to the current example, the loss of precision through floating-point numbers is demonstrated. Compared to the filter example in section 6.1, however, the automatic exponent handling mechanism of floating-point
data types simplyfies computations significantly. Appendix D.5 shows the corresponding hand programmed C-code.

MCS does not necessarily aim to avoid double values for intermediate computations \[6\]. Consequently, by compiling Catalytic’s fixed-point constructs, specifications in method three, described in section 2.4 can not be followed. The closest translation result achieved, is shown in appendix D.4. In this particular case, running this code with the described test data gives in this particular case 29.7812. However, it is hard to base any evaluation on that result, since MCS seems not to be aimed to be used for the generation of target code. This fixed-point code generated by MCS even contains 64-bit integers. A prototype version is available for MCS, which should produce target code: by using for instance the flag -m2c_tiC64x, MCS seems to generate more optimized code. Since it is a prototype, no further investigations have been carried out on this specific compiler mode.

6.3.2 IEEE 802.16d modulator

The bit matrix resulting from the interleaver described in section 3.2 is feed to the modulator. This modulator, for the modulation scheme Quadrature Amplitude Modulation (QAM), generates complex floating-point data. In order to run efficiently on a fixed-point digital signal processor, this modulator needs to be converted to fixed-point. The evaluation of fixed-to-floating-point conversion on this WiMAX modulator for MCS is carried out by following method three in section 2.4. The generation of 16-bit complex fixed-point data is achieved, but the internal utilization of double values could, as for the example given in section 6.3.1 not be avoided.

6.3.3 Filter design facilitation

Catalytic’s fixed-point constructs are applied to certain points of the manual design process. One utilization is at a point when filter coefficients need to be computed by the algorithm itself. This means that the whole computation needs to be translated to fixed-point C. As a result, by the use of MEX to interface the generated code to
MATLAB a SNR of 28.97 dB is achieved. The next point of application is with filter coefficients in double precision pre-calculated, which causes a slight improvement to the SNR of 29.12 dB. Compared to the result of a SNR of 51.57 dB accomplished in section 6.2, the manual translation achieved a much better result. The error print in figure 6.9 shows strong irregularities in the filter behavior.

6.4 The MathWorks target design tools and Embedded MATLAB

Prior to EML, MathWorks already provided fixed-point constructs denoted “Fixed-Point Toolbox” for corresponding simulation purposes. At first glance, this fixed-point tool is somewhat comparable in features and complexity to the RMS add-on.
However, the Fixed-Point Toolbox seems to be more useful to simulate “real” targets, which is exemplified by sections 6.4.1 and 6.4.2. EMLC generates code from Fixed-Point Toolbox, MCS generates code from Catalytic Tools, namely by use of fixed point constructs. A missing feature in MathWorks’ fixed-point environment, however, is an add-on similar to Catalytic’s Profiling Tool as described in section 6.3.

### 6.4.1 The squared average of a vector

In contrast to Catalytic’s solution, the use of Fixed-Point Toolbox enables to specify exactly the maximum sum and product word length [24], as shown in appendix D.1. In that way method three in section 2.4 can be followed. Since the example algorithm in question, consists of the MATLAB function `sum`, Fixed-Point Toolbox recognizes a base 2 logarithmic increase on the required additional data bits to the number of summations [24]. Also, the divide function requiring the desired output numeric type to be specified seems to be a reasonable solution to the fixed-point divide problem [34]. With a result of 33.2812, EML accomplishes better than as a first manual approach, as described in section 6.3.1. Appendix D.3 shows the fixed-point C-code generated by EML.

### 6.4.2 IEEE 802.16d modulator

The Catalytic Tools (as well as EMLC and Fixed-Point Toolbox), can deal with complex numbers in fixed-point representation. The same benchmark setup as described in section 6.3.2 is utilized for investigation on EMLC. Both the MCS and EMLC generated C-code seems to be correct, where as detailed SNR measurements are not undertaken. However, the investigation on the C-code created by EMLC appears to be closer to the target defined for method three in section 2.4 than the version produced by MCS. EMLC does not make use of double precision values and the biggest data type used is a 32-bit integer.
6.4.3 Filter design facilitation

Approaches such as those as described in section 6.3.3 do not show noteworthy better performance with EMLC either. However, MathWorks developed a construct called filter object which for instance covers, a fixed point design process, such as the one described in section 6.1. C-code can be generated together with Simulink & Real-Time Workshop and EMLC. Since the topic of this thesis is MATLAB to C translation, the reason for the investigating these extra tools is to find a way to replace the floating-point MATLAB filter function. This replacement should then be enable to generate a fixed-point filter in C-code by following the specifications for method three in section 2.4.

The filter object can be used from the MATLAB prompt, which results in the following code for the filter design described in section 6.1:

```matlab
%% Design filter
Hf = fdesign.bandpass(’N,Fp1,Fp2,Ap’,8,300,3400,0.5,8000);
n = {’11’,’Linf’,’12’};
filtstruct = {’df1sos’,’df2sos’,’df2tsos’};
reord = {’up’,’down’,’auto’};
s = fdopts.sosscaling;
s.MaxNumerator=10;
s.sosReorder = reord{3};
Hbp = design(Hf,’FilterStructure’,filtstruct{2},
 ’SOSScaleNorm’,n{2},’SOSScaleOpts’,s);

%% Quantize filter
Hbp.Arithmetic = ’fixed’;
Hbp.RoundMode = ’floor’;
Hbp.OutputMode=’BestPrecision’;
Hbp.AccumWordLength = 32;
```

By studying the code, it can be seen that the same choices have been made as for the design in section 6.1. These design choices from section 6.1 are SOS direct-form II, infinity norm scaling and some automatic mechanism is called for section ordering. By setting the filter’s arithmetic to fixed by default 16-bit data types are used by default. What needs to be changed are the accumulator size and some rounding behavior. There is also a graphical way to define this filter object, but for this particular example it does not deliver an equally good SNR. Therefore, the graphical “Filterbuilder” is not further
Figure 6.10: Filter results from the MathWorks’ filter object generated fixed-point C-code compared to the MATLAB floating-point reference filter.
considered. Normally, a Simulink model needs to be designed to produce C-code from an filter object and from this model the C-code is generated. However, Mathworks support provides a script generating C-code automatically by using Simulink in the background.

It is interesting that this C-code, generated from the file object achieves a SNR of 56.66 dB. Although SNR is an important attribute for a filter, there are several more characteristics describing a good filter, such as speed, no limit cycling and stability [19].

Fixed-point filter evaluation is complex area and is beyond the scope of this thesis. The examination of the C-code generated by Real-Time Workshop shows two files, one containing the algorithm and one containing the fixed-point filter coefficients. The length of both files added together, including comments, are 358 lines (approximately 50% are comments). In contrast to the manual filter design results in figure 6.7 the error print in figure 6.10 shows a better behavior. This irregularities disappear if the manual filter is fed with scaled down data. From this point of view it can be assumed that the problem is caused by overflow wrapping. What automatic optimizations MathWorks is additionally applying to the code generated from their filter object, is unknown and not within the scope this thesis.

6.5 Conclusions on fixed-to-floating-point conversion

The discussions in section 1.2 link back to the problem of compiling MATLAB’s floating-point filter function to fixed-point C. The pure MATLAB language does not provide constructs to enter sufficient information in order to, for instance, convert a floating-point filter to a fixed-point filter. The investigation of the manual filter design approach, given in section 6.1 shows that an automation of this manual design process is possible. Going from this initial design automation by increasing experience, a good set of filter design MATLAB scripts can be collected. Further, C libraries interfaced through MEX can be used for simulation in MATLAB and then directly be used for the final target implementation or reference design. Indeed, in a company like Ericsson
AB, such scripts and libraries are used. MathWorks’ filter object can be described as a collection of such scripts, with a common interface and the ability to generate C code. Or from another perspective, a filter object is a M-code annotation to accomplish MATLAB to C-code compiler optimization. However, the evaluation of filter design automation could be a thesis on its own.

The fixed-point constructs of MCS and EMLC for the filter implementation process are not sufficient. However, for some simpler problems automated floating-to-fixed-point conversion can be considered, e.g. for the example given in section 6.4.1. In particular MathWorks’ Fixed Point Toolbox showed promising potential for C-code generation. Catalytic’s fixed-point constructs can not avoid the generation of variables of type double for intermediate computations.
From an Engineer’s Perspective

Software tools should facilitate the engineer’s task in researching for new solutions and to finally design products. As already brought up in chapter 1, nowadays compilers constitute a crucial part of many engineering processes. In chapter 6 the current author called a filter design manual, by still applying MATLAB. Indeed, compared to an automatic MATLAB to fixed-point C translation, the process described in chapter 6 can be considered as manual. However, if viewed from the perspective of having no such tool as MATLAB or the open-source alternative Octave, the approach in chapter 6 may seem less manual. Partly, MATLABs success seems to be due to the fact that it supports design work rather than implying unnecessary overhead. In this thesis investigations on Catalytic Tools and Embedded MATLAB are carried out to give an idea about these compilers’ current abilities.

7.1 Learning curve

Minimum additional overhead by maximum gain in design efficiency could be a description of the perfect engineering aid. The time it takes an engineer to learn a specific tool is overhead and needs to be kept as small as possible. To evaluate a tool on how fast it can be learned would require a certain amount of test people. Further, statistical means would have to be applied to draw meaningful conclusions. The approach in this thesis is more concentrated on a description of the personal experience of the thesis’ author. First, it is the design of the tool itself which determines how fast an
engineer can work efficiently with the design aid. Additionally, documentation and product support are probably crucial factors, for getting acquainted with a new way of working.

Catalytic provides for RMS and MCS, a 2 page Quick Start Guide which helps to carry out: 1) MATLAB to C translation; 2) initial translation optimization; 3) fixed-point simulation; 4) MATLAB simulation speed-up. In order to properly understand how to operate Catalytic tools, a recommended way is to start with a document called Catalytic RMS Overview and Tutorial. MCS is built upon RMS and as a result it is very helpful to spend time in going through these 26 pages of RMS overview and to undertake the tutorial. As MATLAB to C compilation is not a straightforward problem, Catalytic Tools offers several ways to optimize a translation for a particular use. For RMS there is documentation comprising of 196 A4 pages available, and for MCS A4 128 pages. It is not necessary to go through all these pages in detail, but it is useful to understand the full ability of the tools. There the author considers, that in one to two weeks sufficient comprehension should have been built up to be able to use these tools efficiently. As for MATLAB functions description about compiler flags/modes is also available at the MATLAB prompt with the flag -help. Catalytic support is as it can be expected from a business such as Catalytic; accessible and fast.

Next to the tools’ ability, the success of MATLAB and its Toolboxes is probably due to MathWorks’ comprehensive documentation. Beginning with the MATLAB -help flag right available from the prompt till several tutorials available to their Toolboxes, learning is well supported. The Fixed-Point toolbox documentation consist of 181 A4 pages and EMLC documentation goes up to 182 A4 pages. Tutorials are available for both tools and can be recommended as a point of departure. Compared to Catalytic’s documentation and support, Fixed-Point toolbox and EMLC are comparably well supported. For these tools together one or two weeks are estimated to build up enough knowledge of how to apply these design-aids to a certain problem.
7.2 Ease of MATLAB to C translation and usability of generated code

“MATLAB to C translation is not equal MATLAB to C translation.” The generated C programs are used at different design stages and at different targets, by simultaneously having different requirements. For what is referred to as “one click translation” of MATLAB to C, Catalytic Tools seems to be the closest. As the name indicates, Embedded MATLAB is targeting hardware implementation directly from MATLAB code. Therefore a minor subset of MATLAB is supported by EML, which is basically due to the need for generating static code. To give an example: the MATLAB function \textit{dec2bin} can not be supported, owing to the dependability of the output argument’s size on the input argument. To realize a \textit{dec2bin} function as it is in C requires dynamic memory allocation. As a result, the engineer is more constrained by the functions available and in designing algorithms generally.

EML’s strength is “embeddable” code. By following the EML language as described in [23], static and fixed-point C-code of acceptable performance can be produced. As described in 6.4.3, in conjunction to the filter object also a fixed-point filter in C-code is accomplished. The question remains, however, where MCS and EML could be used. Simulink is a different way of engineering than MATLAB and seems to be more suitable to generate C-code from. Today it is possible to write user defined Simulink blocks in EML and generate C-code from the whole Simulink model, which appears to be an useful application of EML. Regarding plain MATLAB to C translation at, in particular, big businesses, there are application designers working with MATLAB, who are unaware of target specifications. In such cases it seems to be hard to have the designer developing directly in EML. According to an engineer at Ericsson AB, often, it is especially the free and interactive way of MATLAB programming, which leads to innovative algorithm design. It probably should not even be the goal of restricting an application designer in any sense, or blocking his/her mind with hardware specifications. However, most of the time, the application designer will know what type and shape the freely developed algorithm’s variables must use. Going from there, by only amending type and shape information to the input arguments of a top-level function and generation
of a algorithm reference in C-code seems to be of use. Hardware verification tools work well with C-code, where dynamic memory allocation is not a problem. Especially when it comes to design iteration, changes could be made directly in the M-code, thereby maintaining the C-reference automatically. Also, having automatically generated C reference code in addition to M-code could facilitate the target implementation process. Consequently, design iteration for target implementation would speed up too. As it supports MATLAB well and requires a minimum of M-code annotations to generate C-code, MCS appears to be the better solution.

There are also companies or design processes that require the same engineer using MATLAB for algorithm design, to later deploy the program to a C target. For such applications, EML could be of interest. EML restricts the use of MATLAB more than MCS does, but EML is still much closer to MATLAB than C is. In that way, only one source would need to be maintained and design iteration would speed up. The deploying from the MATLAB subset Embedded MATLAB directly to target seems to be more recommendable than trying it with MCS. In particular, when it comes to fixed-point implementation, as shown in chapter 6, EML appears to deliver better and more exact results. MCS is not designed to avoid intermediate floating-point computations as shown in section 6.3.1.

7.3 Future outline

Since EMLC and MCS are under ongoing development, results the present author found today can be totally different tomorrow. According to Catalytic Inc. for MCS a functionality is scheduled, to guarantee even for arrays with unknown size static memory allocation. To achieve this static memory allocation, only a maximum size for the array has to be specified. From this point of view the advantage of EMLC to guarantee static code generation could become compromised by MCS. Also for EMLC a similar functionality should be released in the near future to address problems resulting through the dynamic kind of MATLAB, as stated by MathWorks. In that way the problem of the dec2bin function as well as the issue of different modulation schemes as discussed in section 5.5.2 could be solved.
EMLC and MCS translate M-code to C-code in MATLAB’s column-major representation. This requires transposing or in-place transposition of data from external C-code, before the data can be applied to the automatically generated code. Already discussed and illustrated in section 1.2.2, for many embedded applications an additional data transpose is not acceptable. In this respect the automatic generation of “row-major” C-code is crucial for direct target deployment. According to Catalytic Inc. a new release of MCS generates C-code following the row-major data layout to hold matrices. Consequently, external C-code interfaces nicely with automatically generated C-code. However, for EMLC the generation of “row-major” C-code has not been proposed yet.
Conclusion

MATLAB is an interpreted language and therefore, in contrast to C, not designed for compilation. Compiling C-code to assembly code is a complex task and for certain applications it is at present still required to optimize compiler generated assembly code by hand. This thesis describes the basic concepts compilers are built upon. In addition, in order to understand the difficulty of compiling MATLAB to C, interpreters are discussed. MATLAB has several attributes facilitating algorithm development, e.g. the engineer does not need to declare any type, shape or size of variables, resizing of arrays is supported at runtime and the result can be seen after directly entering a statement. The lacking information content in M-code to be interpreted, is gathered from the runtime context. Since no runtime context exists for compilation, several approaches of type, shape and size inference have been researched over the years. Several alternative ways of MATLAB to C compilation are described in the current thesis.

The two compilers Matlab to C Synthesis (MCS) and Embedded MATLAB C (EMLC) are experimented on using three distinct methods: 1) generation of reference code; 2) target code generation; 3) floating-to-fixed-point conversion. Initial investigations on a combined Bubble Sort and Binary Search algorithm are executed; in this respect MCS demonstrates to be more suitable for reference code generation, whereas EMLC delivers better results regarding deployment on target. Both compilers show equally good results on common MATLAB intrinsics. On the other hand, application on a WiMAX signal-processing chain in M-code developed without having automatic MATLAB to C conversion in mind, leads to more difficulties. Where MCS can generate algorithm
references in C straight away for some functions, EMLC does not succeed, basically due to the special needs to generate static code. In particular, build-in functions requiring dynamic memory allocation cannot be supported by EMLC. Generally a lack of functions supported leads to problems for both compilers investigated.

As another example, manual fixed-point filter implementation is shown. A tool enabling fixed-point filter design together with EMLC delivers interesting C-code results. MCS offers a tool to capture and statistically present data distribution on variables of a particular simulation run. From this tool, suggestions are generated for variable quantization. In general EMLC could be better directed to a particular target when it came to fixed-point implementations. MCS appears to be more suitable for reference code generation, by better supporting the MATLAB language as such, whereas EMLC obviously aims for target code generation. The compilers dealt with in the present thesis demonstrated great potential for automatic MATLAB to C translation. In order to guarantee static code generation, later versions of MCS and EMLC probably come with a functionality to declare only the maximum physical size a certain array can have. A new version of MCS is generating “row-major” C-code, which can be crucial for embedded applications. The compiler MCS seems to become in future more suitable also for the generation of target code. On the other hand, the development of EMLC seems to move towards the support of a greater subset of MATLAB. Both tools, EMLC and MCS, have been initially released no longer than a year ago [10, 16]; from this perspective MATLAB to C compilation has a promising future.
Bibliography


List of Abbreviations

ANSI  American National Standards Institute
AST  Abstract Syntax Tree
bit  binary digit
BLAS  Basic Linear Algebra Subprograms
CEO  Chief Executive Officer
CONLAB  Concurrent Laboratory
CPU  Central Processing Unit
dB  deciBel
DSP  Digital Signal Processing
EISPACK  EIgenSystem Package
EML  Embedded MATLAB
EMLC  Embedded MATLAB C
EMLMEX  Embedded MATLAB MEX
FALCON  Fast Array Language COmpilatioN
FFT  Fast Fourier Transform
FFTW  Fastest Fourier Transform in the West
FIR  Finite Impulse Response
FOS  First Order Section
FPGA  Field Programmable Gate Array
FPU  Floating Point Unit
FSB  Front Side Bus
GCC  GNU Compiler Collection
**List of Abbreviations**

**GHz**  Giga Hertz  
**GNU**  GNU’s Not Unix  
**GUI**  Graphical User Interface  
**HDL**  Hardware Description Language  
**HPC**  High Performance Computers  
**Hz**  Hertz  
**IBM**  International Business Machines Corporation  
**IIR**  Infinite Impulse Response  
**JIT**  Just In Time  
**LaPACK**  Linear Algebra PACKage (successor of LINPACK)  
**LINPACK**  Linear Algebra Package  
**M-code**  MATLAB-code  
**MaJIC**  Matlab Just In Time Compiler  
**MATCH**  MATlab Compiler for Heterogeneous systems  
**MATLAB**  MATrix LABoratory  
**MB**  Mega Byte  
**MCC**  MATLAB C Compiler  
**MCS**  Matlab to C Synthesis  
**MEX**  Matlab EXchange  
**MHz**  Mega Hertz  
**MIMD**  Mulitple Instruction Multiple Data  
**MIT**  Massachusetts Institute of Technology  
**MPI**  Message Passing Interface  
**OS**  Operating System  
**PC**  Personal Computer  
**QAM**  Quadrature Amplitude Modulation  
**RAM**  Random Access Memory  
**RMS**  Rapid Matlab Simulator
**RTExpress** Real-Time Express

**RTL** Register Transfer Level

**ScaLAPACK** Scalable Linear Algebra PACKage

**SSA** Static Single Assignment

**SIMD** Single Instruction Multiple Data

**SNR** Signal to quantisation Noise Ratio

**SOS** Second Order Section

**VHDL** Very High Speed Integrated Circuit

**VME** Versa Module Europa

**WiMAX** Worldwide interoperability for Microwave Access
Appendix
Driver to Benchmark C programs

This program has been created to execute benchmarks on floating-point programs as described for method one and method two in sections 2.2 and 2.3.
```c
#include <stdio.h>
#include <stdlib.h>
#include "cc_encode.h"

int* transpose1(int *x, int rows, int cols) {
    int i, j;
    int *x1;

    if ((x1 = (int *) malloc(rows * cols * sizeof(int))) == 0) printf("Memory allocation error!\n\r");

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            x1[rows * j + i] = x[cols * i + j];
        }
    }

    free(x);
    return(x1);
}

int* transpose2(int *x, int cols, int rows) {
    int i, j;
    int *x1;

    if ((x1 = (int *) malloc(rows * cols * sizeof(int))) == 0) printf("Memory allocation error!\n\r");

    for (i = 0; i < cols; i++) {
        for (j = 0; j < rows; j++) {
            x1[j * cols + i] = x[rows * i + j];
        }
    }

    free(x);
    return(x1);
}

int main(int argc, char** argv) {
    /*for file read*/
    const char DATA_FILE_YCC[] = "YRS10000.txt";
    FILE *fileYCC;
    char line[80];

    /*for time measurement*/
    double stTime, endTime, exTime;
```
struct timeval tv;

int i, j;
int *YCC; /* holding burst blocks*/
double rate = 0.5;
int iTemp;

/*return values*/
int *y;
int y_dim1, y_dim2;

/*read in matrix YCC from file*/
fileYCC = fopen(DATA_FILE_YCC, "r");
if (fileYCC == NULL) {
    fprintf(stderr, "Error: Unable to open %s\n", DATA_FILE_YCC);
    exit (8);
}

/*count items in file*/
for (i = 0; 1; i++) {
    if (fgets(line, sizeof(line),
        fileYCC) == NULL)
        break;
}

/*allocate space for items in file*/
if ((YCC = (int *) malloc(i * sizeof(int))) == 0)
    printf("Memory allocation error!\n\r");
if ((y = (int *) malloc(192 * 10000 * sizeof(int)))
    == 0) printf("Memory allocation error!\n\r");
rewind (fileYCC);

/*write data to array*/
for (i = 0; 1; i++) {
    if (fgets(line, sizeof(line),
        fileYCC) == NULL)
        break;
    sscanf(line, "%d", &iTemp);
    YCC[i] = (int) iTemp;
}
/*end read in matrix YCC*/

/*check read in from file, by displaying first column*/
for(i = 0; i < 12; i++)
    for (j = 0; j < 1; j++) {
A. Driver to Benchmark C programs

```c
95  printf("\%d", YCC[10000 * i + j]);
96  fflush(stdout);
97  
98  /*transpose to MATLAB column-major*/
99  YCC = transpose1(YCC, 12, 10000);
100
gmtime(&tv, NULL);
101  stTime = tv.tv_sec * 1000000 + tv.tv_usec;
102
cce_decode(YCC, rate, y);
103
gmtime(&tv, NULL);
104  endTime = tv.tv_sec * 1000000 + tv.tv_usec;
105  exTime = (endTime - stTime) / 1000000;
106  printf("\nExecution time was: \%f seconds. \n", exTime);
107
transpose back to C row-major/
108  y = transpose2(y, 10000, 192);
109
tprint out last column to quick check correct result
110  for (i = 0; i < 192; i++) {
111      for (j = 9999; j < 10000; j++) {
112          printf("\%d ", (int) y[10000 * i + j]);
113      }
114  }
115  printf("\n");
116  
117  free(YCC);
118  free(y);
119  return 0;
120  }  /*print out last column to quick check correct result
121  for (i = 0; i < 192; i++) {
122      for (j = 9999; j < 10000; j++) {
123          printf("\%d ", (int) y[10000 * i + j]);
124      }
125  }
126  */
```
B

Initial Investigation: Sort and Search

The Bubble Sort and the Binary Search algorithm are used to illustrate first translation results from EMLC and MCS compared to a manual translation.
B. Initial Investigation: Sort and Search

B.1 Original MATLAB source

```matlab
function middle = sortSearch(M, key)

% Initialize array
N = size(M, 2);

% Bubble sort
for i = N-1:-1:0
    test = 1;
    for j = 1:i
        if M(j) > M(j+1)
            test = 0;
            temp = M(j);
            M(j) = M(j+1);
            M(j+1) = temp;
        end
    end
const test == 1
    break;
end

% Binary search
low = 1;
high = length(M);
while low <= high
    middle = ceil((low + high) / 2);
    if key == M(middle)
        return;
    end
    elseif key < M(middle)
        high = middle -1;
    else
        low = middle + 1;
    end
end
middle = -1;
```
B.2 Translation to C-code by hand

```c
int binarySearch(int a[], int key, int length) {
    int low = 0;
    int high = length - 1; /*zero based array*/
    int middle;

    while (low <= high) {
        middle = ceil((low + high) / 2);
        if (key == a[middle]) /*match*/
            return middle + 1;
        else if (key < a[middle]) /*search low end of array*/
            high = middle - 1;
        else /*search high end of array*/
            low = middle + 1;
    }

    return -1; /*search key not found*/
}

void bubbleSort (int x[], int n) {
    int i, j;
    int tmp;
    int test; /*test if already sorted.*/

    for (i = n - 1; i > 0; i--) {
        test = 1;
        for (j = 0; j < i; j++) {
            if (x[j] > x[j+1]) {
                test = 0;
                tmp = x[j];
                x[j] = x[j+1];
                x[j+1] = tmp;
            }
        }
        if(test)
            break;
    }
}
```
B. Initial Investigation: Sort and Search

B.3 Translation result of Catalytic Tools

```c
#ifndef CT_MCS_MEX
#include "ct_target_m2c.h"
#else
#include "ct_target_mex.h"
#endif

#include "ct_runtime.h"

#include "sortSearch.h"

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

FUNCTION: sortSearch
/***********************************************************************************/

double _sortSearch(int32 M[10000], int32 key) {
    int32 (*M_1)[10000];
    double middle_out, low, high;
    int32 test, temp, i1, i2;

    CT_INIT(M_1);
    CT_ALLOC(M_1, 1, int32 (*)[10000]);
    for (i1=0; i1<9999; i1+=1) {
        (*M_1)[i1] = M[i1];
    }
    for (i1=9999; i1>0; i1+=-1) {
        test = 1;
        for (i2=0; i2<=i1 - 1; i2+=1) {
            if ( (*M_1)[i2] > (*M_1)[1 + i2] ) {
                test = 0;
            }
        }
    }
    middle_out = temp;
    return middle_out;
}
```
B. Initial Investigation: Sort and Search

```
46        temp = (*M_1)[i2];
47        (*M_1)[i2] = (*M_1)[1 + i2];
48        (*M_1)[1 + i2] = temp;
49    }
50    }
51    if (test == 1) {
52        break;
53    }
54 }
55    low = 1.0;
56    high = 10000.0;
57    while (low <= high) {
58        middle_out = ceil((low + high) / 2.0);
59        if (key == (*M_1)[((int32)middle_out) - 1]) {
60            goto _THE_END_;
61        }
62        else if (key < (*M_1)[((int32)middle_out) - 1]) {
63            high = middle_out - 1.0;
64        }
65        else {
66            low = 1.0 + middle_out;
67        }
68    }
69    middle_out = -1.0;
70    _THE_END_;;
71    CT_FREE(M_1);
72    return middle_out;
73 }
```
B.4 Translation result of Embedded Matlab

```c
/* sortSearch.c */
/* Embedded MATLAB Coder code generation for M-function */
/* 'sortSearch' */
/* C source code generated on: Mon Dec 3 00:06:12 2007 */
*/

/* Include files */
#include "sortSearch.h"

/* Type Definitions */
/* Variable Declarations */
/* Variable Definitions */
/* Function Declarations */
/* Function Definitions */

real_T sortSearch(int32_T *eml_M, int32_T eml_key)
{
    real_T eml_middle;
    int32_T eml_i;
    int32_T eml_exitg2;
    int32_T eml_test;
    int32_T eml_j;
    int32_T eml_temp;

    static int16_T eml_iv0[2] = { 1, 10000 };;
    int32_T eml_exitg1;
    /* Initialize array */
    /* Bubble sort */
    eml_i = 9999;

    do {
    eml_exitg2 = 0U;
    if(eml_i >= 0) {
        eml_test = 1;
        for(eml_j = 1; eml_j <= eml_i; eml_j++) {
            if(eml_M[eml_j - 1] > eml_M[eml_j]) {
                eml_test = 0;
                eml_temp = eml_M[eml_j - 1];
                eml_M[eml_j - 1] = eml_M[eml_j];
                eml_M[eml_j] = eml_temp;
            }
        }
    }
    }"
if (eml_test == 1) {
    eml_exitg2 = 1U;
} else {
    eml_i += -1;
}
}
else {
    eml_exitg2 = 1U;
}
} while (eml_exitg2 == 0U);

/* Binary search */
eml_i = 1;
eml_test = 0;
for (eml_j = 0; eml_j < 2; eml_j++) {
    if (eml_iv0[eml_j] > eml_test) {
        eml_test = (int32_T)eml_iv0[eml_j];
    }
}
do {
    eml_exitg1 = 0U;
    if (eml_i <= eml_test) {
        eml_middle = ceil((real_T)(eml_i + eml_test) / 2.0);
        if (eml_key == eml_M[(int32_T)eml_middle - 1]) {
            eml_exitg1 = 1U;
        } else if (eml_key < eml_M[(int32_T)eml_middle - 1]) {
            eml_test = (int32_T)eml_middle - 1;
        } else {
            eml_i = (int32_T)eml_middle + 1;
        }
    } else {
        eml_middle = -1.0;
        eml_exitg1 = 1U;
    }
} while (eml_exitg1 == 0U);
return eml_middle;

void sortSearch_initialize(void)
{
    rt_InitInfAndNaN(8U);
}

void sortSearch_terminate(void)
{
}

/* End of Embedded MATLAB Coder code generation
(sortSearch.c) */
Manual fixed-point filter in C

The algorithm in this appendix constitutes a manual IIR filter fixed-point implementation in C. Furthermore, a MEX interface construct is used to connect this program to the MATLAB prompt.
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define X_IN prhs[0]
/* Output Arguments */
#define Y_OUT plhs[0]

#if !defined(MAX)
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#endif

#if !defined(MIN)
#define MIN(A, B) ((A) < (B) ? (A) : (B))
#endif

void fixFilt(short *x, short *b, short *a, short q) {
    short w[3] = {0, 0, 0};
    short n;
    long accu;
    for (n = 0; n < 1000; n++) {
        accu = a[1] * w[1];
        accu = accu >> q;
        w[0] = x[n] - accu;
        accu = a[2] * w[2];
        accu = accu >> q;
        w[0] = w[0] - accu;
        x[n] = accu >> q;
        w[2] = w[1];
        w[1] = w[0];
    }
}

void filtInf(double *y, double *x) {
    short b1[3] = {13239, -26478, 13240};
    short a1[3] = {2048, -3848, 1903};
    short b2[3] = {9684, -19367, 9683};
    short a2[3] = {8192, -12062, 4897};
    short b3[3] = {1352, 2705, 1353};
    short a3[3] = {16384, 27420, 14198};
    short b4[3] = {3056, 6113, 3056};
    short a4[3] = {16384, 16042, 6135};
    short x16[1000];
    long accu;
    short i;
    for (i = 0; i < 1000; i++) {
        x16[i] = (short)(x[i] * pow(2, 15));
        accu = x16[i] * 25910;
        x16[i] = accu >> 16;
C. Manual fixed-point filter in C

48    }  
49    fixFilt(x16, b4, a4, 14);  
50    fixFilt(x16, b2, a2, 13);  
51    fixFilt(x16, b3, a3, 14);  
52    fixFilt(x16, b1, a1, 11);  
53    for (i = 0; i < 1000; i++) {
54      y[i] = (double)(x16[i] * pow(2, -15));  
55      y[i] = y[i] * 7.1897534256509; /* tilde values */
56    }
57
58  
59  void mexFunction(int nlhs, mxArray *plhs[], int nrhs,  
60               const mxArray*prhs[]) {
61    double *y,*x;  
62    mwSize m,n;  
63 /* Check for proper number of arguments */
64    if (nrhs != 1) {
65      mexErrMsgTxt("One input argument required.");
66    } else if (nlhs > 1) {
67      mexErrMsgTxt("Too many output arguments.");
68    }
69 /* Check the dimensions of X. X can be 1000 X 1  
70 or 1 X 1000. */
71    m = mxGetM(X_IN);  
72    n = mxGetN(X_IN);  
73    if (!mxIsDouble(X_IN) || mxIsComplex(X_IN) || (MAX(m,n)  
74        != 1000) || (MIN(m,n) != 1)) {
75      mexErrMsgTxt("filtCasInf requires that X be  
76 a 1000x1000 vector.");
77    }
78 /* Create a matrix for the return argument */
79    Y_OUT = mxCreateDoubleMatrix(1, 1000, mxREAL);  
80 /* Assign pointers to the various parameters */
81    y = mxGetPr(Y_OUT);  
82    x = mxGetPr(X_IN);  
83 /* Do the actual computations in a subroutine */
84    filtInf(y,x);  
85  }
Squared average of a vector computation in fixed-point

The squared average of a vector calculation in fixed-point should provide an initial view on how to use the fixed-point constructs of MCS and EMLC. In addition, the corresponding translations compared to a manual translation, are illustrated.
D. Squared average of a vector computation in fixed-point

D.1 Fixed-point EML in MATLAB

```matlab
function y = sum2EML(x)
assert(isa(x,'double') & isreal(x) &
   all(size(x) == [1 1999]));
F = fimath('RoundMode','floor','OverflowMode','wrap',
   'ProductMode','KeepMSB','ProductWordLength',32,
   'SumMode','KeepMSB','SumWordLength',32,
   'CastBeforeSum',true);
Tdiv = numerictype('Signed',false,'WordLength',16,
   'FractionLength',6);
fx = fi(x,1,16,6,F);
a = fi(fx.^2,0,16,7,F);
b = sum(a);
l = fi(length(x),0,16,4,F);
y = divide(Tdiv,b,l);
```

D.2 Fixed-point MCS in MATLAB

```matlab
function y = sum2MCS(x)
fxp_init('sw','m','f',0,0,0,'prop_off');
fxp_accum_guard(0);
fxp_short_mode('on');
mbfxprow(10,6,'s',x);mbsize([1 1999],x);
a = x.^2;
b = sum(a);
l = fxp(length(x),12,4);
y = b/l;
```

D.3 Fixed-point C from EML

```c
/*
 * sum2EML.c
 *
 * Embedded MATLAB Coder code generation for
 * M-function 'sum2EML'
 *
 * C source code generated on: Sun Dec 9 19:28:34 2007
 */

* Include files */
#include "sum2EML.h"

/** Type Definitions */
/** Variable Declarations */
/** Variable Definitions */
/** Function Declarations */
static void m_power(int16_T *eml_a, real_T eml_b, int16_T *eml_y);
static uint32_T m_sum(uint16_T *eml_X);
/** Function Definitions */
uint16_T sum2EML(real_T *eml_x) {
    int32_T eml_i0;
    real_T eml_d0;
    int16_T eml_iv0[1999];
    int16_T eml_iv1[1999];
    uint16_T eml_uv0[1999];
    for(eml_i0 = 0; eml_i0 < 1999; eml_i0++) {
        eml_d0 = fmod(floor(ldexp(eml_x[eml_i0], 6)), 65536.0);
        if(eml_d0 < -32768.0) {
            eml_d0 += 65536.0;
        } else if(eml_d0 >= 32768.0) {
            eml_d0 -= 65536.0;
        }
    }
    eml_iv0[eml_i0] = (int16_T)eml_d0;
    m_power((int16_T *)&eml_iv0, 2.0, (int16_T *)&eml_iv1);
    for(eml_i0 = 0; eml_i0 < 1999; eml_i0++) {
        eml_uv0[eml_i0] = (uint16_T)((uint16_T)
            eml_iv1[eml_i0] << 1);
    }
    return (uint16_T)(uint32_T)((0 ? MAX_uint32_T : (uint32_T)(m_sum((uint16_T *)eml_uv0) / 31984U)) >> 2);
}

void sum2EML_initialize(void) {
    rt_InitInfAndNaN(8U);
}

void sum2EML_terminate(void) {
static void m_power(int16_T *eml_a, real_T eml_b,
        int16_T *eml_y)
{
    int32_T eml_i1;
    real_T eml_d1;
    for(eml_i1 = 0; eml_i1 < 1999; eml_i1++) {
        eml_y[eml_i1] = 0;
        eml_d1 = fmod(floor(ldexp(pow(ldexp(
                      (real_T)eml_a[eml_i1], -6), eml_b), 6)),
                          65536.0);
        if(eml_d1 < -32768.0) {
            eml_d1 += 65536.0;
        } else if(eml_d1 >= 32768.0) {
            eml_d1 -= 65536.0;
        }
        eml_y[eml_i1] = (int16_T)eml_d1;
    }
}

static uint32_T m_sum(uint16_T *eml_X)
{
    uint32_T eml_Y;
    int32_T eml_k;
    eml_Y = (uint32_T)eml_X[0] << 5;
    for(eml_k = 2; eml_k < 2000; eml_k++) {
        eml_Y += (uint32_T)eml_X[eml_k - 1] << 5;
    }
    return eml_Y;
}

/* End of Embedded MATLAB Coder code generation (sum2EML.c) */

D.4 Fixed-point C from MCS

/* End User Licensee may distribute C code generated from */
/* Catalytic licensed functions only in combination with */
/* the original work of Licensee. */
/* Catalytic MCS-file info */
/* Created : Sun Dec 9 16:01:12 2007 */
#ifndef CT_MCS_MEX
#include "ct_target_m2c.h"
#else
#include "ct_target_mex.h"
#endif

#include "ct_runtime.h"

#include "sum2MCS.h"

// sum2MCS.m:1 function y = sum2MCS(x)

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

FUNCTION: _sum2MCS

**************************************************************************
int16 _sum2MCS(int16 x[1999]) {
    int16 y_out; /* fxp(8,8) */
    int16 b, stemp_0; /* fxp(19,-3) */
    int32 itemp_0; /* fxp(27,-3) */
    int32 i;

    // sum2MCS.m:2 fxp_init ('sw', 'm', 'f', 0,
    // 0, 0, 0, 'prop_off ');
    // sum2MCS.m:3 fxp_accum_guard (8);  
    // sum2MCS.m:4 fxp_short_mode ('on ');
    // sum2MCS.m:5 mbfxprow(10, 6, 's', x);
    // mbsize([1 1999], x);
    // sum2MCS.m:7 a = x.ˆ2;
    // sum2MCS.m:8 b = sum(a);
    itemp_0 = 0;
    for (i=0; i<=1998; i+=1) {
        stemp_0 = (int16) RND_FLOOR(((int32) x[i]) *
        ((int32) x[i]), 0, 15);
        itemp_0 = ADD_MOD(itemp_0, FXP_EXTEND(((int32)
        stemp_0), 8, int32), 8, int32);
    }
    b = ((int16) itemp_0);
    // sum2MCS.m:9 l = fxp(length(x),12,4);
    // sum2MCS.m:10 y = b/l;
    y_out = (int16) RND_ZERO((int64) (b << 16) /
D. Squared average of a vector computation in fixed-point

57  ((int32) 31984)), 0x2, 1, 0x1);
58  return y_out;
59 }

D.5 Hand coded floating- and fixed-point C

1 #include <stdio.h>
2 #include <math.h>
3 // typedef for PC
4 typedef signed char S8;
5 typedef unsigned char U8;
6 typedef signed short S16;
7 typedef unsigned short U16;
8 typedef signed long S32;
9 typedef unsigned long U32;
10
11 float sum2(float *x) {
12    float a[1000], b, y;
13    U16 i;
14    for (i = 0; i < 1000; i++)
15        a[i] = x[i] * x[i];
16    for (i = 0; i < 1000; i++)
17        b += a[i];
18    y = b/1000;
19    return y;
20 }
21
22 float* sum2Fp(float *x) {
23    S16 q9_6_x[1999];
24    U32 q18_12_a[1999];
25    U16 q10_6_a[1999];
26    U32 q26_6_b;
27    U32 q18_12_b;
28    U16 q17_m1_b; // ufxp(17, -1)
29    U32 q17_15_d_16; // 16 bit sum
30    U32 q17_15_d_32; // 32 bit sum
31    float y[2];
32    U16 i;
33    //mapping to fxp
34    for (i = 0; i < 1999; i++)
35        q9_6_x[i] = round(x[i] * pow(2, 6));
36    printf("1:%d\n", q9_6_x[0] >> 6);
37    //quadratation
38    for (i = 0; i < 1999; i++)
39        q18_12_a[i] = q9_6_x[i] * q9_6_x[i];
40    printf("2:%d\n", q18_12_a[0] >> 12);
// shift right and put into 16 bit
for (i = 0; i < 1999; i++)
    q10_6_a[i] = q18_12_a[i] >> 6;
printf("3: %d\n", q10_6_a[0] >> 6);
// compute sum from 16 bit with 32 bit accumulator
q26_6_b = 0;
for (i = 0; i < 1999; i++)
    q26_6_b += q10_6_a[i];
printf("4: %d\n", q26_6_b >> 6);
// shift left and put back into 16 bit
q17_m1_b = q10_6_a[i] << 7;
printf("6: %d\n", q17_m1_b << 1);
// compute sum from 32 bit and keep 32 bit
q18_12_b = 0;
for (i = 0; i < 1999; i++)
    q18_12_b += q18_12_a[i];
printf("5: %d\n", q18_12_b >> 12);
// division with 32 bit, accumulate in 16 bit
q17_15_d_16 = (q17_m1_b << 16) / ((U32) 1999);
printf("7: %d\n", q17_15_d_16 >> 15);
// division with 32 bit, accumulate in 32 bit
q17_15_d_32 = (q18_12_b << 3) / ((U32) 1999);
printf("8: %d\n", q17_15_d_32 >> 15);
// remapping results to float
y[0] = q17_15_d_16 * pow(2, -15);
y[1] = q17_15_d_32 * pow(2, -15);
return y;