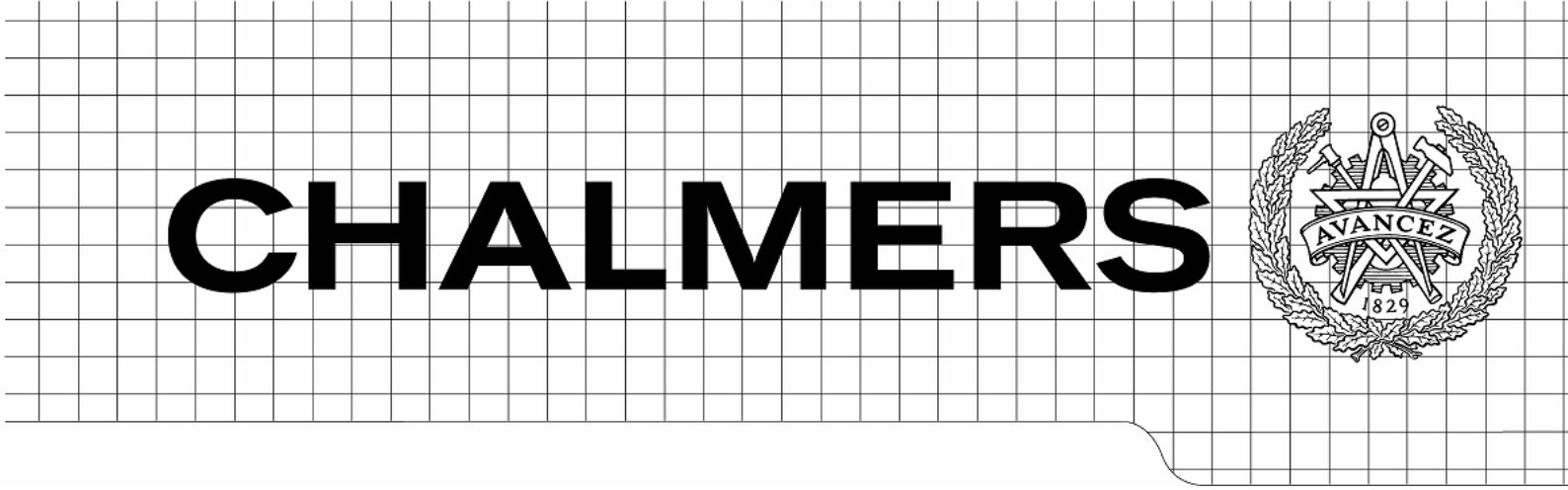

Generating Embedded C Code for Digital Signal Processing

Master of Science Thesis in Computer Science - Algorithms, Languages and Logic

Mats Nyrenius
David Ramström

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

Generating Embedded C Code for Digital Signal Processing

Mats Nyrenius
David Ramström

Examiner: Mary Sheeran
Supervisor at Chalmers: Emil Axelsson
Supervisors at Ericsson AB: Peter Brauer and Henrik Sahlin

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2011

Abstract

C code generation from high-level languages is an area of increasing interest. This is because manual translation from specifications to C code is both time consuming and prone to errors. In this thesis, the functional language Feldspar has been compared to MATLAB (together with MATLAB Coder) in terms of productivity and performance of generated code.

Several test programs were implemented in both languages to reveal possible differences. The set of test programs included both small programs, testing very specific properties, as well as more realistic digital signal processing algorithms for mobile communications. The generated code was run on two different platforms: an ordinary PC and a Texas Instruments C6670 simulator. Execution time and memory consumption were evaluated. For the productivity evaluation, four different areas important to software development were defined. This was followed by reasoning about the languages in each area, using test programs as examples.

The results show that MATLAB generally performs better. The problems of Feldspar observed in this thesis, however, are limited to a few details which should be possible to improve. Also, in some cases Feldspar performed better because of an optimization technique called fusion. The productivity evaluation showed some interesting differences between the languages, for instance regarding readability and type safety.

Acknowledgements

We would like to thank the Baseband Research group at Ericsson AB for welcoming us to your inspiring work environment. Special thanks go to our supervisors Peter Brauer and Henrik Sahlin. Your help and technical expertise has been of great importance during this thesis.

The same goes for Emil Axelsson, who was our supervisor at Chalmers, and Anders Persson. When programming in Feldspar, your help with reviewing code and explaining language details has been invaluable.

We would also like to thank Fredrik Rodin at MathWorks for helping us a lot with MATLAB. You have helped us by reviewing our code and report, as well as by giving us technical advice.

Finally, we would like to thank our examiner, Mary Sheeran, for all help with the report.

Contents

List of Abbreviations

3GPP	3rd Generation Partnership Project
ASIC	Application-Specific Integrated Circuit
CD	Compact Disc
ChEst	Channel Estimation
DRS	Demodulated Reference Symbols
DRY	Don't Repeat Yourself
DSP	Digital Signal Processing
DSPs	Digital Signal Processors
EML	Embedded MATLAB
Feldspar	Functional Embedded Language for DSP and Parallelism
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
GHC	The Glasgow Haskell Compiler
IDE	Integrated Development Environment
IFFT	Inverse Fast Fourier Transform
LTE	Long Term Evolution
MATLAB	Matrix Laboratory
MMSE EQ	Minimum Mean Square Error Equalization
PC	Personal Computer
SAC	Single Assignment C
TI	Texas Instruments
TIC64x	TMS320C64x Chip Family
TIC6670	TMS320C6670

Chapter 1

Introduction

1.1 Background

Digital signal processing (DSP) is the analysis and manipulation of signals in digital form. Usually, these signals are physical impressions from the real world, such as sound waves and images. Since there is no way to fully represent the measured data from such a continuous analogue signal in a computer, one has to sample it into discrete values of time and amplitude to get a digital representation. This is done using an analogue to digital converter. Then the processing can take place, which usually involves filtering, measuring or compression. The signal can then be converted back to an analogue signal using a digital to analogue converter.

DSP has its origin in the 1960's when digital computers became available. These were very expensive at the time, which made the use of DSP limited to military and governmental applications such as radar, sonar and seismic oil exploration. In the 1980's and 1990's when personal computers became publicly available, the DSP area broadened significantly and suddenly it became used in all kinds of commercial applications. Some examples from then until today are voice mail, CD players, mobile phones, digital music equipment, image processing software and digital television.

DSP algorithms are run on various platforms, often on standard computers, but also frequently on specialized microprocessors, digital signal processors (DSPs). When speed is a major concern, field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC) might be used.

In digital signal processing software today, most code is written in low level C, highly optimized for the specific target platform, due to performance demands. High performance is usually crucial, since many applications, especially in communications, have real-time demands. Writing the algorithms in C usually works well, but when an application is to be moved to a different target platform, it might be necessary to rewrite the entire code because of all the optimizations. This is time consuming and error prone, which makes it a very expensive task. Also, the C code is not very intuitive to read because of its machine-oriented nature. Heavy use of intrinsic instructions and other adaptions for specific hardware makes it hard to read and maintain.

The demands on DSP software will probably increase rapidly over the next few years. A reason for this is the never ending pursuit of higher bit-rates in mobile communications. In order to keep up with the demands, extensive use of parallel hardware and software is needed. It is hard and error prone to write such code in C, since the programmer manually has to take care of all details

concerning parallelization.

Instead, an intuitive and maintainable high-level language would be preferred for writing the DSP algorithms. This language could then be compiled into target code for different platforms, preferably with multi-core support, and without sacrificing performance. However, most DSP targets today only have C compilers, and it is standard to write the software in C. Efficient code generation from a high-level language to C code is thus a subject which seems to be of increasing interest. Below, a few examples of such products are briefly discussed.

Feldspar [1] is a high-level functional programming language targeting DSP applications. Since it is a functional language, programs in Feldspar usually follow a data-flow programming style [2], meaning that computations are described as networks of data structure transformations. The language is a work in progress with objectives of increasing maintainability and portability without sacrificing performance. Currently, there is an ISO C99 back-end available, as well as limited support for the TIC64x family, but the intention is to provide back-ends for many DSPs in the future.

MATLAB (MATrix LABoratory) [3] is a numerical computing environment and high-level programming language focused around vector and matrix manipulations. It features a large amount of built-in functions for many kinds of mathematical calculations, and also has built-in visualization functionalities for easy data analysis.

MATLAB Coder [4] is a tool for generating C code from MATLAB code, which supports code generation for many different target platforms. A subset of the MATLAB language is supported for C code generation [5], including most functionalities needed for algorithm design. This subset together with a compiler for C code generation was referred to as Embedded MATLAB (EML) until MATLAB 2011a.

Single Assignment C (SAC) [6] is, like Feldspar, an ongoing project, with design goals that could be attractive for DSP applications. For instance, high-level representations of multi-dimensional arrays and array operations are stated objectives of the language. Another goal is compilation for parallel program execution in multiprocessor environments. SAC, however, does not have specific focus on code generation for embedded systems, like Feldspar and MATLAB Coder.

1.2 Related Work

One question is if these languages satisfy the desired properties of being portable and maintainable, while still providing performance comparable to handwritten C code. There has been some work made on evaluating the Embedded MATLAB C compiler, where the generated C code was analyzed to evaluate readability and modularity in order to see if EML could contribute to faster and more efficient development processes in a company [7]. The results showed that it would contribute to the development process when generating floating-point reference C code. The reference code could be used as reference when writing optimized code for certain hardware. However, generation of fixed-point C code was considered to not contribute to the development process.

Also, a comparison between Embedded MATLAB and Catalytic MCS has been made in [8]. As one might expect, the results shows some advantages and disadvantages of each product in different areas. MCS proved to be better at reference code generation, covering a larger set of the MATLAB language, while EML seemed more suitable for direct target implementation. It should be noted that Catalytic is today owned by MathWorks.

1.3 Purpose

The results of this thesis are intended to give Ericsson and the developers of both Feldspar and MATLAB insight in how the languages perform against each other, and to point out areas in need of improvement. The results will be useful for Ericsson when considering using automatic C code generation from high-level languages in the DSP algorithm design process.

The main focus has been on Feldspar, since Ericsson wanted to know how far it had come in its development. However, MATLAB is widely used at Ericsson which makes its results important as well.

1.4 Method

In order to fulfill the purpose, an extensive comparison between Feldspar and MATLAB was made. The main aspects of the languages to evaluate were productivity and performance of generated C code.

The performance, referred to as hard measures, was evaluated by measuring clock cycles and memory usage for a set of test programs. The precise method of doing this is explained in detail in section 4.1. The test programs include small programs designed to expose specific weaknesses in the languages, to yield a good base for comparison. Larger test programs implementing DSP algorithms were also used for evaluating the performance in realistic situations.

For evaluating the productivity, also referred to as soft measures, a number of areas were chosen according to section 1.6. These areas were defined based on scientific research and personal thoughts, and in the evaluation the languages were reasoned about with respect to the definitions.

A survey was also composed, with the purpose of getting opinions from people with varying experience.

1.5 Objectives

An objective of this thesis was to pinpoint eventual advantages and disadvantages of the languages in various areas, in order to give relevant feedback to the language developers. Another objective was to give Ericsson insight in Feldspar's current status. Also, an objective was to produce platform independent results.

The success of the project can be measured by how well differences in performance and productivity are explained, assuming there are any.

1.6 Delimitations

1.6.1 Soft Measures

There are of course many interesting areas which could have been included in the soft measures, but because of the limited time frame of the project, four areas were chosen as specified in section 5.1.

The soft measures mainly concerned the high-level code, except for readability, where the generated C code was also taken into account. There, such things as variable naming and unnecessary code were considered.

1.6.2 Fixed-Point Arithmetic and Multi-Core Support

The Feldspar version used in the beginning of this thesis had very limited support for fixed-point arithmetic, only supporting simple arithmetic expressions. Even though MATLAB has support for fixed-point arithmetic, it was not evaluated since no comparison could be made.

Neither of the languages currently support generation of parallel C code, which is why this was not part of the evaluation. However, possibilities for multi-core support are briefly discussed in sections 2.1.7 and 2.2.5.

1.6.3 Reference Code

The hard measures evaluation is only a comparison between Feldspar and MATLAB. There is no handwritten reference code, so it is important to not draw the conclusion that the best result is the same as a good result compared to the ideal case.

1.6.4 Correctness of Generated C Code

Correctness for the code generation will be assumed, and thus not tested very much in the hard measures. In other words the generated C code is assumed to yield the same results as the high-level code. It is expected that the language developers have made certain of this.

1.6.5 Intrinsics

The C code was generated to not include any platform specific intrinsics, because Feldspar has no support for this for the TIC6670 which was used in this thesis. Feldspar currently only supports generating code according to the ISO C99 standard, and the TIC64x family to some extent, which has no floating-point support. Since no fixed-point arithmetic was used in the thesis, it was necessary to use a platform supporting floating-point arithmetic.

1.6.6 Memory

Memory is usually allocated statically in embedded systems. MATLAB Coder is able to dynamically allocate memory, but because of the reason mentioned, and since Feldspar currently cannot generate code with dynamic memory allocation, only static allocation was used in this thesis.

1.6.7 Survey

Since there are few persons who have actually seen Feldspar, it would be hard to make the survey very scientific. Because of this, the results from the survey were not used to draw any conclusions, but rather to get some general opinions about Feldspar and MATLAB.

1.6.8 Single Assignment C

Single Assignment C seemed like an interesting language to include in the evaluation. However, after some initial research it was observed that in order to compile the generated C code, a number of libraries were required. These were only available as precompiled libraries for Linux x86, and since other platforms were used in this thesis, SAC was not evaluated.

1.7 Report Structure

This section describes the structure of the report to give the reader an overview of the content.

Chapter 1 contains a background and purpose motivating the thesis. The objectives, method and delimitations are also stated here.

Chapter 2 introduces the two languages Feldspar and MATLAB, and provides theory used in the forthcoming chapters.

Chapter 3 describes and motivates all test programs used in the evaluation.

Chapter 4 contains the performance evaluation of the languages. The method used for this is stated, together with encountered problems. Finally, the results are presented and explained.

Chapter 5 contains the productivity evaluation. Areas important to productivity are first defined, and then both languages are evaluated in the defined areas.

Chapter 6 contains a discussion concerning the results of the hard and soft measures, as well as differences observed when using the languages.

Chapter 7 contains a short description of Feldspar's current status, and feedback to the developers of Feldspar and MATLAB. A section describing possibilites of future work related to this thesis is also put here.

Finally, there are appendices containing all high-level source code, tables of execution time results and the survey together with answers.

Chapter 2

Languages

2.1 Feldspar

2.1.1 Introduction

Feldspar (Functional Embedded Language for DSP and PARallelism) is a domain specific language embedded in the functional programming language Haskell which aims for being used for programming DSP algorithms. It is a joint research project between Ericsson AB, Chalmers University of Technology (Göteborg, Sweden) and Eötvös Loránd University (Budapest, Hungary).

Feldspar comes with an associated code generator, currently only supporting ISO C99 code generation, which have been used in this thesis. There is also limited support for the TMS320C64x chip family, and a future goal is to support many different DSPs and FPGA platforms.

Programming in Feldspar is designed to be as much as programming in Haskell as possible. For example, most of Haskell's list library is implemented in Feldspar's vector library described in section 2.1.3. Feldspar is built around a core language, which is a purely functional language on a level of abstraction similar to C. The idea is to build various libraries upon this core language, and that these libraries should provide a higher level of abstraction for programming.

As mentioned, Feldspar is purely functional, meaning that it is guaranteed that no side effects can occur. A function is said to have side effects, if it other than resulting in a value, also in some way gives an effect outside the function. Examples of side effects are change of global variables and printing output to the display or a file. Because there are no side effects, Feldspar is referentially transparent, which means that a function always yield the same output on the same input. Features such as higher order functions, anonymous functions and polymorphism are inherited from Haskell. Feldspar also has the same static and strong type system as Haskell (see section 5.3.1).

In section 2.1.2 to 2.1.5, some Feldspar features used in this thesis are briefly described. In section 2.1.6, the limited support for fixed-point arithmetic is described, and in section 2.1.7 possibilities for generating multi-core code are discussed.

2.1.2 Working with Feldspar

Feldspar is imported as an ordinary library in Haskell, which makes it very convenient for programmers familiar with Haskell. In this thesis, the Glasgow Haskell Compiler (GHC) version 6.12.3 [9] is used, together with the interpreter GHCi. This version was supplied with the Haskell Platform, version 2010.2.0.0 [10]. The Feldspar version used in this thesis was 0.4.0.2, but since it was not released when the project started, different development revisions were used at first. To use Feldspar, import it by writing `import Feldspar` in the top of a Haskell source file.

A Feldspar program has the type `Data a`, where `a` is the type of the value computed by the program. Consider the following example of a function `add`, which takes two arguments of type `Data Int32` and returns a value of type `Data Int32`:

```
import Feldspar

add :: Data Int32 -> Data Int32 -> Data Int32
add a b = a + b
```

To evaluate the function `add`, the `eval` function can be used:

```
*Main> eval (add 1 2)
3
```

Notice that since Feldspar is strongly typed, it is not possible to apply `add` to arguments of any types other than `Int32`.

The `eval` function makes it possible to easily verify that functions are doing what they are expected to do. However, the Feldspar developers have not yet put much effort in making it fast, so for heavy computations `eval` is very slow.

Core Language

The programmer probably wants to use the libraries in order to program at a high level of abstraction similar to Haskell. However, it is also possible to program directly in the core language, which gives more in-depth control over the generated C code.

The function `value` may be used to convert a Haskell value into a Feldspar value. It may also be used to convert a Haskell list into a Feldspar core array:

```
*Main> eval (value [1..10 :: Int32])
[1,2,3,4,5,6,7,8,9,10]
```

The core language includes a function for computing the elements in a core array independently from each other, called `parallel`.

```
parallel :: (Type a) => Data Length -> (Data Index -> Data a) -> Data [a]
```

The function `parallel` takes two arguments. The first one is the length of the array, and the second is a function which computes the element at a given index. The `parallel` function is interesting for multi-core applications, as briefly discussed in section 2.1.7.

Partial Function Application

When applying a function, it is not necessary to supply all its arguments. If applying `add` to just one argument of type `Data Int32`, this will return a function which takes one argument of type `Data Int32` and returns a value of type `Data Int32`.

```
(add 2) :: Data Int32 -> Data Int32
```

`(add 2)` returns a function which takes one argument of type `Data Int32` and adds `2` to it. This lets us simplify the first `add` function as:

```
add :: Data Int32 -> Data Int32 -> Data Int32
add = (+)
```

Polymorphism

It is often the case that one wants to use the same function for different input and output types. For example, it is likely that one wants an `add` function for floating-point numbers as well. This can be accomplished using Haskell's polymorphism, and below follows an example of an polymorphic version of `add`:

```
add :: Num a => Data a -> Data a -> Data a
add = (+)
```

`Num a` means that `a` must be an instance of the class `Num`, which requires that each its members must implement `(+)` amongst some other functions. See [11] for more information.

Anonymous Functions

In Feldspar, it is not necessary to bind a function to an identifier. An anonymous function may be defined and used in the following way:

```
*Main> eval $ (\a b -> ((a+b)::Data Float)/2) 1 2
1.5
```

This defines a function which takes two arguments, `a` and `b`, and computes the average of them. The `$` operator is used for avoiding parantheses. Anything after it will take precedence over anything that comes before.

Anonymous functions are very convenient when working with higher-order functions, which are functions that take a function as argument and/or return a function.

Loops

One fundamental difference when programming in Feldspar, compared to Haskell, is that recursion on Feldspar values is not allowed. In Haskell, recursion is the usual way to accomplish looping, so this might be confusing for a programmer used to Haskell. Instead, there are special functions such as `forLoop`:

```
forLoop
:: (Syntactic st) =>
    Data Length -> st -> (Data Index -> st -> st) -> st
```

The first argument is the number of iterations. A limitation of the `forLoop` function is that there is no way to break the iterations. `forLoop` will do exactly as many iterations as specified in the first argument. The second argument is the starting state, which has to be a member of the class `Syntactic` (explained below). The last argument is a function, which takes an index and the current state, and computes the next state. The final state is the value which is returned by the function.

If a type `a` is `Syntactic`, it basically means that there is a way to go from `a` to `Data b` (for some `b`) and back, without changing the semantics. For example, a Haskell pair of Feldspar values `(Data a, Data a)` can be converted to and from a Feldspar pair `Data (a, a)`. This can be very useful because it enables the programmer to use many of Haskell's nice features, such as pattern matching [12] See the *BinarySearch* test program for an example of this, where a Haskell pair is used as the state of `forLoop`.

The following example shows a function which sums the elements of a vector:

```
sumVec :: DVector Int32 -> Data Int32
sumVec v = forLoop (length v) 0 (\i st -> st + v!i)
```

2.1.3 Libraries

Feldspar comes with a number of libraries to make it possible to program at a higher level of abstraction than the core language. In this thesis, the vector and matrix libraries described below are the libraries that have been used the most.

Vector Library

The majority of functions in Haskell's list library are implemented for Feldspar vectors in the vector library. Programmers familiar with Haskell will find programming with Feldspar's vector library very similiar to programming with Haskell lists.

The vector library is perhaps the most important library in Feldspar. A difference between vectors and core arrays is that vectors does not allocate any memory in the generated C code unless the programmer explicitly forces this. A vector is described by a size and an indexing function, which computes the element at a given index. The example below shows a vector of length 10 and an indexing function which multiplies the index with 2, or in other words, a vector containing the elements $0, 2, 4, 6, 8, 10, 12, 14, 16, 18$.

```
import Feldspar.Vector

vec :: DVector DefaultWord
vec = indexed 10 (\i -> i*2)
```

It is possible to convert a Haskell list to a vector using the function `vector`.

```
vector :: (Type a) => [a] -> Vector (Data a)
```

An example where a vector is used without allocating any memory for it is presented below, as a modified version of `sumVec`.

```
sumVec :: Data DefaultWord
sumVec = forLoop (length vec) 0 (\i st -> st + vec!i)
```

`sumVec` no longer takes a vector as input. Instead, it uses the vector from the previous example. The example compiles to the following C code.

```
void sumVec(struct array mem, uint32_t * out0)
{
    uint32_t temp1;

    (* out0) = 0;
    {
        uint32_t i2;
        for(i2 = 0; i2 < 10; i2 += 1)
        {
            temp1 = ((* out0) + (i2 << 1));
            (* out0) = temp1;
        }
    }
}
```

As seen, no memory is allocated for `vec`. Its elements are instead added directly to the sum. The first argument `mem` is decribed in section 2.1.5.

Operations on vectors can be built up in a very compositional style without sacrificing performance, thanks to an optimization technique called fusion which is described in section 2.1.4.

Matrix Library

A Feldspar matrix is just short for a vector of vectors, so it is possible to use many functions from the vector library on matrices as well. The matrix library currently only implements very basic matrix operations, like matrix multiplication, transpose and functions for converting between core matrices (core array of core arrays) and indexed matrices (vector of vectors). Functions for computing matrix inverse and determinants are absent, which made the implementation for the *MMSE EQ* test program in section 3.3.4 difficult.

To use the matrix library, the module `Feldspar.Matrix` has to be imported.

2.1.4 Fusion

One of the most important features of the vector library (section 2.1.3) is an optimization method called fusion [1] [13]. Fusion for vectors guarantees that no intermediate vectors between computations in a program will be stored. This is perhaps best illustrated by an example, which is shown below.

```
dotRev :: DVector Int32 -> Data Int32
dotRev v = scalarProd v $ reverse v
```

`dotRev` computes the scalar product of a vector and its reverse. This compiles to the following C code:

```
void dotRev(struct array mem, struct array in0, int32_t * out1)
{
    uint32_t v2;
    uint32_t v3;
    int32_t temp4;

    v2 = length(in0);
    v3 = (v2 - 1);
    (* out1) = 0;
    {
        uint32_t i5;
        for(i5 = 0; i5 < v2; i5 += 1)
        {
            temp4 = ((* out1) + (at(int32_t,in0,i5) * at(int32_t,in0,(v3 - i5)))
);
            (* out1) = temp4;
        }
    }
}
```

This leads to two very important observations. Firstly, there is no need to store the result of the first function (`reverse`) in an intermediate array, which would have resulted in higher memory consumption. Secondly, without fusion, there would have been two for-loops, one for reversing the vector, and one for the scalar product. Thus fusion is likely to improve execution time, even though the improvement might differ depending on the platform.

However, fusion might not always be the desired behaviour. The programmer may choose to explicitly force allocation of an intermediate vector by using the function `force`. Below follows an example of `dotRev` without fusion, and the corresponding generated C code:

```
dotRev :: DVector Int32 -> Data Int32
dotRev v = scalarProd v $ force $ reverse v
```

C code:

```
void dotRev(struct array mem, struct array in0, int32_t * out1)
```

```
{
    uint32_t v2;
    uint32_t v4;
    int32_t temp6;

    v2 = length(in0);
    v4 = (v2 - 1);
    setLength(&at(struct array,mem,0), v2);
    {
        uint32_t i5;
        for(i5 = 0; i5 < v2; i5 += 1)
        {
            at(int32_t,at(struct array,mem,0),i5) = at(int32_t,in0,(v4 - i5));
        }
    }
    (* out1) = 0;
    {
        uint32_t i7;
        for(i7 = 0; i7 < v2; i7 += 1)
        {
            temp6 = ((* out1) + (at(int32_t,in0,i7) * at(int32_t,at(struct array,mem,0),i7)));
            (* out1) = temp6;
        }
    }
}
```

As seen, there are now two for-loops, and the result of the first computation is stored in an intermediate array in `mem`. See section 2.1.5 for more information about memory handling.

The test program *TwoFir* (see section 3.2.2) gives an example where it might be good to trade off memory consumption for avoiding repeated computations.

2.1.5 C Code Generation

Compiling

To compile a Feldspar function, the first step is to import the `Feldspar.Compiler` module. The `compile` function can then be used to compile a Feldspar function:

```
Main*> compile sqAvg "sqAvg.c" "sqAvg" defaultOptions
```

The first argument is the Feldspar function to compile, the second is the name of the output file, the third is the name of the C function and the last contains compiler options.

The function `icompile` can be used to return the generated code to the Haskell interpreter instead.

There are four different predefined compiler options:

- `defaultOptions`

Generate C code according to the ISO C99 standard. All compilation steps are performed and no loop unrolling is made.

- `tic64xPlatformOptions`
  Generate C code compatible with the Texas Instruments TMS320C64 chip family. All compilation steps are performed and no loop unrolling is made.
- `unrollOptions`
  All compilation steps are performed and innermost loops are unrolled at most 8 times.
- `noPrimitiveInstructionHandling`
  Turn on the NoPrimitiveInstructionHandling debugging option [14]

See the user's guide to the Feldspar compiler for more details [14].

For this thesis, a custom set of compiler options was made due to problems regarding complex numbers, as explained in section 4.5.1.

Memory

Arrays in the C code are represented by an array structure, `struct array`, containing three fields. The first field is the length of the array, the second is a pointer to a buffer containing the data, and the third is the size of an element. For a nested array, the last field is set to `-1`.

The `at` macro is used to access the array structures, and takes three arguments. The first argument is the type of the elements in the array, the second is the array structure, and the third is the index.

All memory handling is up to the programmer to take care of. The special array structure `mem`, which is always the first argument to the generated C function, is a nested array structure which contains pointers to all memory that the function needs. Currently, the programmer has to manually analyze the C code to figure out the size and structure of `mem`, and then allocate the necessary memory. This proved to be a very tedious task throughout the project (see section 4.5.3).

Input Vectors of Explicit Length

It is possible to explicitly set the length of an input vector, which can sometimes help the compiler to generate better C code. This can be done using the built-in function `wrap`, as demonstrated by the following example:

```
twoFir :: DVector Float -> DVector Float -> DVector Float -> DVector Float
twoFir b1 b2 = convolution b1 . convolution b2

twoFir_wrap
    :: Data' D10 [Float]
    -> Data' D10 [Float]
    -> Data' D100 [Float]
    -> Data [Float]
twoFir_wrap = wrap twoFir
```

The example above shows the the test program *TwoFir* (see section 3.2.2) used with `wrap`. `Data'` is an extension of `Data` for use with wrappers to explicitly set input vector lengths. The lengths are denoted by a `D` followed by the desired length.

A problem with `wrap` is that it does not currently support multi-dimensional vectors of arbitrary lengths. However, it is possible to set explicit input lengths in a more manual way by using the function `unfreezeVector'` which is the method used in this thesis referred to as *wrap*.

```
unfreezeVector' :: Type a => Length -> Data [a] -> Vector (Data a)
```

`unfreezeVector` takes a length and a core array, and returns a vector. Note that the length is a Haskell value, which means that the length will be fixed inside the Feldspar program. The programmer can define functions using vectors as usual, and then define a separate wrapper function to be used for compilation. The following example shows a wrapper function for `dotRev`, which takes a vector of size `1024` as input:

```
dotRev_wrap :: Data [Int32] -> Data Int32
dotRev_wrap v = dotRev (unfreezeVector' 1024 v)
```

This generates the following code:

```
void test(struct array mem, struct array in0, int32_t * out1)
{
    int32_t temp2;

    (* out1) = 0;
    {
        uint32_t i3;
        for(i3 = 0; i3 < 1024; i3 += 1)
        {
            temp2 = ((* out1) + (at(int32_t,in0,i3) * at(int32_t,in0,(1023 - i3)
)));
            (* out1) = temp2;
        }
    }
}
```

As seen, the C code is now slightly more static. The number of iterations in the for-loop is now fixed.

2.1.6 Fixed-Point Arithmetic

Fixed-point data types are used to describe fractional numbers using a fixed range and precision, in contrast to floating-point data types, which can have varying range and precision [15]. Fixed-point data types are useful in situations where floating-point arithmetic is not supported, which is the case in many embedded systems.

The development version of Feldspar used in the beginning of this thesis had almost no support for fixed-point arithmetic. Fixed-point numbers could not be used together with most language

features, like the `forLoop` function amongst others. The only thing available was basic arithmetic like addition and multiplication. No interesting comparison could thus be made with MATLAB, which has thorough support for fixed-point arithmetic (see section 2.2.4).

The current release of Feldspar has more support for fixed-point arithmetic, but it is still not fully developed. See the Feldspar tutorial [2] for more information.

2.1.7 Multi-Core Support

Feldspar's core language contains constructs for expressing arrays in which the elements can be calculated independently of the others. One such construct is `parallel`, which in theory allows the compiler to generate C code for calculating the elements of the array in parallel. The current Feldspar compiler does not support generation of parallel C code, but this is a future goal.

Another thing that bodes well for generation of multi-core C code from Feldspar is that it does not allow side effects, which means that the compiler is free to run parts of the program in any order - or in parallel - as long as data dependencies are met.

2.2 MATLAB

2.2.1 Introduction

MATLAB is an integrated development environment (IDE) and a high-level programming language focused around matrices and matrix operations. It is a product developed by MathWorks [16].

MATLAB originates from two Fortran libraries developed in the 1970's, namely Eispack and Linpack, which included features for calculating matrix eigenvalues and solving linear equations. The first version of MATLAB was implemented in Fortran using portions of these libraries, and the only data type available was "matrix" [17].

Cleve Moler, co-author of Linpack and Eispack and author of this first MATLAB version, taught a numerical analysis course at Stanford University in the late 1970's in which the students were to use MATLAB for some of the homework assignments. The students came from many different backgrounds, and it was the students from engineering which seemed to appreciate the emphasis on matrices the most. Coding with matrices and matrix operations was useful for them because they studied areas like control analysis and signal processing, where matrices play a central role.

Since the 70's, MATLAB has evolved into the full-fledged IDE and high-level programming language it is today. The early indications at Stanford University of MATLAB's usefulness for engineering proved to be accurate. Today MATLAB is used extensively in industry areas like signal processing, image processing, mathematics, medical engineering and research etc [18].

In signal processing for instance, MATLAB is often used for designing and testing algorithms, which are later going to be deployed onto embedded systems. The algorithms usually have to be reimplemented by hand to the target language, which can be a long and tedious process for large systems. It is however possible to compile MATLAB programs to C code, which is the main functionality in MATLAB examined in this thesis.

MATLAB Coder [5] is the tool used for generating C code from MATLAB code. A subset of

the MATLAB language is supported for code generation, which includes functionalities typically used by engineers for developing algorithms, with the exception of visualization functionalities. All MATLAB attributes and functions supported for code generation can be found in the user's guide [4]. The subset of MATLAB supported for code generation was earlier referred to as Embedded MATLAB.

One of the objectives of MATLAB Coder is to generate code with high readability, which is an aspect that will be examined in chapter 5.

In the following sections, some general information about working in MATLAB is presented, then code generation using MATLAB coder is described, and finally fixed point arithmetic support and the possibilities of generating code for multi-core settings are briefly discussed.

2.2.2 Working with MATLAB

Environment

One of MATLAB's key features is the command line, where the user can enter calculations line by line and store values in variables. The variables are stored in the MATLAB workspace and can be saved for use in other MATLAB sessions. The example below shows how a matrix multiplication can be computed using the command line.

```
>> X = [1 2; 3 4];
>> Y = [4 3; 2 1];
>> X*Y

ans =

    8     5
    20    13

>>
```

In the first two lines, the `2x2` matrices `X` and `Y` are defined. Then they are multiplied in the third line and the result is shown. The workspace now contains the variables `X`, `Y` and `ans`, where the result is stored.

The user may write series of MATLAB commands in files called scripts. Scripts can be run from the command line, executing the commands inside line by line. The user may also create functions which can be used in the command line or in scripts. An example function is shown later in this section.

It should be noted that MATLAB is an object-oriented language. See [19] for more information on this.

Matrix Arithmetic for Performance

Programming in MATLAB is preferably done using vector and matrix arithmetic where possible, because it often yields higher performance than writing C like code. This can involve reformulating

problems to use vectors and matrices instead of, for instance, for-loops [20].

Functions and Types

In MATLAB, there is no need to be explicit about the input and output types of a function. A function works as long as its sub-routines do not produce any run-time errors for the used input types. This is discussed in more detail in section 5.3.1.

The built-in functions of MATLAB are often overloaded to work with different scalar types as well as vectors and matrices. For instance, if a function for operating on scalars has been implemented, it will instantly work for vectors and matrices as long as the sub-routines used in the function work accordingly. This is the case in the function shown below:

```
function x = loopadd(x,k)
    for i=1:k
       x = x*2 + i;
    end
end
```

The function `loopadd` computes `x = ((x*2 + 1)*2 + 2)*2 + 3 ...` until the rightmost term reaches `k`. The operators `+` and `*` are defined both for scalars and matrices. For scalars, the operators work as intended. If `x` is a matrix, `x*2 + i` is element-wise multiplication of `2` and then element-wise addition of `i`. Thus the function works both for scalars and matrices.

Constraints can be put on the input and output types using the function `assert` [21], and when generating C code from a function, the input and output types should be specified to achieve desired behaviour of the C code.

2.2.3 MATLAB Coder

C Code Generation

In this thesis, MATLAB 2011a was used, which includes MATLAB Coder. The following steps describe how to generate C code from a MATLAB function using MATLAB Coder. The first step is required since it enables the user to run compiled C code as *MEX functions* from the MATLAB environment, which is part of the recommended workflow for using MATLAB Coder. See [4] for more information on this.

- Select a C compiler for use with MATLAB by entering the command `mex -setup`.
- Add the directive `%#codegen` after the function declaration to enable code generation error checking.
- Make sure that the function to generate C code from does not use any features not supported by MATLAB Coder, such as visualization functions. If there are unsupported features in the code when compiling it, an error will be produced.
- Create and modify a build configuration object as desired

```
>> cfg = coder.config('exe');
>> open cfg
```

- Enter the the following command and provide example input arguments of desired types to the -args flag

```
>> codegen -config cfg -args {...} function_name
```

A new folder is created in the current folder in MATLAB, containing the generated C code, and a report file is generated if the option is enabled in the build configuration file.

The following example illustrates how MATLAB Coder is used. The function `loopadd` described in section 2.2.2 is compiled to C code using the codegen command:

```
>> codegen -c -config cfg -args {zeros(1,100000,'int32'),int32(100)} loopadd
```

The build configuration object `cfg` used for the test programs in this thesis is found in appendix A.5. The first example argument provided to the `-args` flag is a vector of integers of length `100000`, and the second is the integer `100`. Resulting C code:

```
void loopadd(int32_T x[100000], int32_T k)
{
    int32_T i;
    int32_T i0;

    for (i = 1; i <= k; i++) {
        for (i0 = 0; i0 < 100000; i0++) {
             x[i0] = (x[i0] << 1) + i;
        }
    }
}
```

Note that MATLAB Coder does not allow C code generation of MATLAB functions containing type errors (which is allowed for MATLAB functions as mentioned in section 2.2.2).

Memory

MATLAB Coder supports three methods of memory allocation:

- Static memory allocation with hard-coded sizes in the generated C code.
- Static memory allocation with size variables provided when calling the generated C function. Code is generated for checking index bounds.
- Dynamic memory allocation.

MATLAB Coder uses static memory allocation by default, but the desired method can be selected in the build setting configuration (see [4]). An upper limit for stack usage can also be set. If this limit is reached, the program starts allocating memory on the heap.

2.2.4 Fixed-Point Arithmetic

An explanation of fixed-point arithmetic is found in section 2.1.6

The Fixed-Point Toolbox for MATLAB [15] enables the user to program using fixed-point data types and arithmetic, and it is also possible to generate fixed-point MATLAB code to C code using MATLAB Coder. However, fixed-point arithmetic was not evaluated in this thesis because of limitations in Feldspar, as explained in section 1.6.2.

2.2.5 Multi-Core Support

The Parallel Computing Toolbox [22] for MATLAB lets the user divide the workload of algorithms over available processor cores as desired. The toolbox includes for instance the `parfor` loop, which works like a regular for-loop except that each iteration is done independently of the others, dividing the workload over available processor cores. It should be noted that execution of `parfor` does not guarantee deterministic results [23].

A webinar held by MathWorks about the Parallel Computing Toolbox was attended by the authors of this thesis, who raised the question if there were any plans on supporting generation of parallel C code using this toolbox and MATLAB Coder. The answer was that the need for code-generation for parallel architectures had been identified, and the topic was currently under investigation. This means that such use might be supported in the future. However because generation of parallel C code is currently not supported, no further investigation on the topic is made in this thesis, as stated in section 1.6.2.

Chapter 3

Test Programs

3.1 Introduction

This chapter describes all test programs which were used for the hard and soft measures. The description contains a motivation of why each test is an interesting example, and also how it was implemented in both languages.

For some programs, there are more than one version. In the first version, performance was not taken into account during implementation, and no optimizations were made. The other versions are implementations where details about how the languages and compiler works are taken into account in order to generate better C code. These include Feldspar versions named *wrap*, which denotes that the lengths of the input vectors have been explicitly set, as described in section 2.1.5.

To make the evaluation fair, all Feldspar test programs were sent to a Feldspar developer for review, and all MATLAB test programs were sent to a developer at MathWorks for review. The feedback was either used to make small fixes, or to make new optimized versions.

3.2 Small Test Programs

3.2.1 Motivation

The small test programs are supposed to expose certain strengths and weaknesses in Feldspar and MATLAB and their compilers for C code generation. The test programs were not arbitrarily chosen; the areas to test were found by reading about the languages and talking to the Feldspar developers, in order to get ideas of interesting things to test. All examples might not be of interest to everyone, because they test specific details in the languages. However, it should be useful to have some clues about how the languages perform in various situations.

All implementations of the small test programs are found in appendix A.1.

3.2.2 Test Programs

AddSub

AddSub adds 1 to all elements of a vector, and then subtracts 1 from them. The hypothesis was that the compilers might be able to figure out that nothing has to be done. The MATLAB implementation uses the `+` and `-` operators which are overloaded for use with vectors/matrices and Feldspar uses the `map` function together with `+` and `-`.

BinarySearch

BinarySearch performs binary search on the input vector and the element to search for. If found, its index is returned, otherwise the vector's length +1 is returned. The Feldspar implementation uses the function `forLoop`, which does not allow breaking. The example is made to point this out by comparing it to the MATLAB implementation which can break.

MathWorks had some comments which involved using a more efficient function for integer division and rounding. The original function was changed according to the comments.

BitRev

For each element in the input vector, *BitRev* reverses the bits in the index resulting in a new index, and swaps the elements at these indices. This test program is supposed to show how good the languages handle bit arithmetic. This example was provided as an example in the Feldspar tutorial [2].

BubbleSort

Bubblesort sorts the input vector using the bubble sort algorithm. In MATLAB, this can be written as it would have been in C. In Feldspar however, the vector to sort has to be in the state of a `forLoop` function, which is not what one wants to do in Feldspar (see section 4.5.2). Also, `forLoop` cannot break in Feldspar, which means it has to run the maximum number of iterations, even if the sorting is finished. This test program is supposed to show how well the languages perform when it comes to vector updating.

MathWorks optimized the initial implementation of this example, and the new version uses a while loop both for checking if elements have been swapped and to keep track of the counter variable. Both the original implementation (*BubbleSort1*) and the updated (*BubbleSort opt*) were kept.

This was also a good test program for comparing element swapping between the vector library and the core language (described in section 2.1.2) for the Feldspar implementation; thus two implementations were made. *BubbleSort1* uses the vector library, and *BubbleSort2* uses core arrays.

DotRev

DotRev calculates the dot product between a vector and its reverse. The compilers might be able to fuse the reversing into the dot product calculation, which makes it an interesting example. Both Feldspar and MATLAB have built-in functions for dot product and reverse, but in MATLAB the algorithm can also be optimized by writing C like code. The latter version was made by MathWorks and there are thus two MATLAB versions and one Feldspar version of the example. *DotRev* uses the built-in functions and *DotRev opt* is the optimizied MATLAB implementation.

Duplicate

Duplicate takes a vector as input and returns it appended to itself. There are two implementations in Feldspar to compare if the built-in concatenation function is the best choice (*Duplicate*), or if one could make it better by writing two elements at the same time using a for-loop and core arrays (*Duplicate2*).

MathWorks had no comment on this test program, so there is only one MATLAB implementation.

IdMatrix

IdMatrix takes an integer input N and returns the identity matrix of dimension $N \times N$. In Feldspar, this is an indexed matrix which has 1 as element where the indices are equal. In MATLAB, the built in function `eye` is used.

RevRev

RevRev computes the reverse of the reverse of the input vector. It is interesting to see if this is optimized to do nothing by any of the compilers, which of course is the desired behaviour.

SliceMatrix

SliceMatrix takes two pairs of indices and a parameter N and returns the part of the identity matrix of size $N \times N$ which is in between the indices. The idea was to see if Feldspar could benefit from the fact that fusion makes it possible to only compute the elements inside the desired slice, instead of first computing the matrix and then slice it.

SqAvg

SqAvg computes the squared average of a vector. This is supposed to show if both Feldspar and MATLAB can fuse the squaring with the summation. For MATLAB, there is an optimized version *SqAvg opt*, which computes the sum of squares by a matrix multiplication of the input vector and its transpose.

TwoFir

TwoFir applies two FIR filters in series onto a signal. It takes a vector representing the signal, and two vectors representing the filter coefficients. In MATLAB, the function `filter` is used, and in Feldspar, `convolution` which comes with Feldspar as an example. This example is supposed to show how you can compose many filters, and whether Feldspar or MATLAB generates the best code for this. For Feldspar, another version *TwoFir wrap* was implemented with input vectors of explicit lengths (see section 2.1.5), and not using fusion in order to avoid repeated computations.

TransTrans

TransTrans computes the transpose of the transpose of a matrix A, namely A. This example is supposed to test if the compilers can figure out that nothing has to be done. A function for matrix transpose comes with both Feldspar and MATLAB.

3.3 DSP Test Programs

3.3.1 Introduction

Long Term Evolution (LTE) is the latest standard in mobile network technology, and is developed by the 3rd Generation Partnership Project (3GPP), which is a collaboration between manufacturers of mobile communications all over the world.

The LTE standard defines how the different parts of a mobile network should behave. Below are some key problems of data transmissions in mobile networks described, along with their solutions according to the LTE standard.

To enable base stations to communicate with multiple users, the frequency spectrum is divided into *sub-carriers*, which are distributed among the users. The base station then transmits data to each user in their assigned sub-carriers.

A problem affecting wireless data transmissions in mobile networks is that the signals get affected by reflections on mountains and other surfaces. This is the channel of the transmissions. Also, unwanted noise is added to the signal which needs to be minimized at the receiver to retrieve the sent data correctly. The following model shows how a signal is altered during transmission and methods for extracting it at the receiver are then described. This description is very brief and is only supposed to give the basic picture.

$S(m)$ is the transmitted reference signal, where m is the sub-carrier index.

Received signal model:

$$Y(m) = H(m)S(m) + V(m)$$

where $H(m)$ is the channel and $V(m)$ is noise.

The channel can be estimated using a *reference signal*, known both to the sender and the receiver.

The base station sends a reference signal, $S(m)$ in the model above, which is known at the receiver and can thus be removed by calculating a matched filter channel estimate $H_{est}(m)$.

$$H_{est}(m) = S^*(m)Y(m)$$

where $S^*(m)$ is the complex conjugate of $S(m)$ and $H_{est}(m)$ is the estimated channel. The noise term $V(m)$ can be reduced by transforming the channel estimate into time domain, windowing it and transforming it back.

The channel estimation is used as input to a *Minimum Mean Square Error Equalization (MMSE EQ)* algorithm, which is applied to received signal after the channel estimation. The purpose of this equalization is to minimize the effects of the channel and noise.

The reference signal is generated by the algorithm described in section 3.3.2, the channel estimation is done by the algorithm described in section 3.3.3 and the MMSE EQ algorithm is described in section 3.3.4. All of these algorithms are implemented as test programs used for the hard measures.

3.3.2 Demodulation Reference Symbols (DRS)

General

This example is mainly intended to test how well the languages can handle bit arithmetic, as well as basic complex number arithmetic.

Input:

N_{Sc}	Number of sub-carriers
v	Base sequence number
N_{Drs}	Demodulated reference signal number
C_s	Cyclic shift field number
$Cell_{id}$	Cell id number
$Delta_{ss}$	Parameter configured by higher layers

The only parameter that really makes a difference in the test run, is the number of sub-carriers, since it decides the size of the output vector. For details, see 3GPP 36.211 [24], section 5.5.2.1.1.

Output:

Drs	Vector of demodulation reference symbols in frequency domain, for 20 slots. A slot is a way to refer to a time interval in a transmitted signal.

The reference symbols are generated according to the 3GPP 36.211 technical specification [24]. A significant part of the algorithm is the generation of a pseudo-random sequence of bits, and this is also a place where the two languages' implementations differ. The pseudo-random sequence is used to calculate a cyclic shift which is then used to calculate the demodulation reference symbols. See 3GPP 36.211 section 7.2 for more details.

Feldspar Implementation

Ericsson provided a Feldspar version of the algorithm implemented in an older version of Feldspar than used in this thesis. This Feldspar implementation had been used to generate C code to be run on a platform without floating point arithmetic, and made use of some integer tricks specifically adapted for the platform. In this thesis, the algorithm was modified to be compliant with the current Feldspar version, and to use floating-point arithmetic since that is supported by the target platforms used for testing.

The generation of the pseudo-random sequence, mentioned above, uses the `forLoop` function for computing the bit-sequence. Because it is bad for the performance to use a vector in the state of the `forLoop` function in Feldspar (see section 4.5.2), the provided implementation avoided this by representing the bit vector as a single integer instead. Bit operations were then used to calculate the sequence. This implementation is named *DRS opt*.

A more naive version, *DRS*, was also implemented, where the bit-sequence is represented by a vector of integers. This vector is then used in the state of `forLoop` as discussed above.

MATLAB Implementation

An implementation of *DRS* in MATLAB was supplied by Ericsson. The original implementation calculated the reference symbols for the first slot, this was modified to be made for all 20 slots instead. The implementation was sent to the MathWorks to get feedback on the code, and they pointed out that a for-loop could replace a vector operation in one place, which resulted in slightly better C code. This version is referred to as *DRS*.

The part of the algorithm calculating the pseudo-random sequence relies heavily on looping over vectors containing the bits of the sequence. In the original implementation, the bit sequence was represented by a vector of 0s and 1s. In the optimzied Feldspar version, a different approach was taken as described above. This approach was taken in the optimized MATLAB implementation, *DRS opt*, as well.

The original implementation of the pseudo-random sequenced relied on a function which given an integer, converted it to a vector of 0s and 1s, representing the integer in binary format. This function was not compliant with MATLAB coder, and had to be reimplemented resulting in the function `int2binlist`.

3.3.3 Channel Estimation (ChEst)

General

This test program aims to test how the languages perform against each other when it comes to transforming back and forth between time and frequency domain using FFT and IFFT. It should be noted that MATLAB's functions probably are very well tested and optimized because of its commercial value and widely use for a long period of time, while the functions used for the Feldspar test program are just provided as examples in the language.

It should also be noted that in reality, FFTs and IFFTs are often calculated efficiently using hardware accelerators, but this is still an interesting test program to evaluate because transforms

play a central role in DSP applications.

Input:

H	Matched filter channel estimate constructed by $Y(m) * S^*(m)$
N_{Sc}	Number of sub-carriers

Output:

H_t	Estimated channel H in time domain
H_f	Estimated channel in frequency domain.

ChEst takes two arguments, namely a matched filter channel estimate in frequency domain and the number of sub-carriers. It transforms H to time domain using the inverse fast fourier transform (IFFT). Now, the desired part will be in the beginning of the signal, and the remaining part is only noise which should be removed. A short window containing the channel without noise is cut out and is then transformed back to frequency domain using the fast fourier transform (FFT).

Feldspar Implementation

The implementation in Feldspar was straight forward. Even though this test program does not contain any matrices, element-wise operations from the matrix library were used to make the code simpler (instead of using `map`). Most necessary functions could be found in the libraries, except from `nextpow2`, which computes the closest above power of two of a given integer. This was needed since the `fft` function requires the input vector to be of a length that is a power of two. An optimized version with input vectors of explicit lengths, as described in section 2.1.5, was also made.

MATLAB Implementation

Two MATLAB implementations were supplied by Ericsson, and only minor modifications had to be made in order to make them compliant with MATLAB Coder. It had to be made explicit that the window should contain complex numbers. Also, when using MATLAB Coder, the length of the vector used as argument to `fft` and `ifft` has to be a power of two, which was achieved using the built in function `nextpow2`.

MathWorks had no comments for this implementation.

3.3.4 Minimum Mean Square Error Equalization (MMSE EQ)

General

This test program aims to test how the languages perform when making a large number of matrix calculations.

Input:

H	Channel matrix. $N_{Rx} \times N_{Tx} \times N_{Sc}$ where N_{Rx} is the number of receiving

	antennas, N_{Tx} the number of MIMO streams and N_{Sc} the number of sub-carriers.
C	Noise covariance matrix $N_{Rx} \times N_{Rx}$

Output:

W	Equalization matrix $N_{Rx} \times N_{Tx} \times N_{Sc}$
H_{post}	Equalized channel matrix $N_{Tx} \times N_{Tx} \times N_{Sc}$

The *MMSE EQ* function computes the equalization matrix and equalized channel matrix for all sub-carriers according to:

$$W = H' * (H * H' + C)^{-1} \quad \text{(MMSE EQ1)}$$
$$H_{post} = W * H$$

where H' is the complex conjugate transpose of H.

$$W = (H' * C^{-1} * H + I_{N_T x})^{-1} * H' * C^{-1} \quad \text{(MMSE EQ2)}$$
$$H_{post} = W * H$$

where $I_{N_T x}$ is the identity matrix of size $N_{Tx} \times N_{Tx}$.

MMSE EQ2 might look strange, since there are three matrix inversions. However, the inverse of C only has to be computed once, since C is the same for all sub-carriers. Also, since N_{Tx} is smaller than N_{Rx}, the outer matrix to invert is also small. This is why this method is preferred when N_{Tx} is smaller than N_{Rx}.

Feldspar Implementation

The main problem with the Feldspar implementation was that Feldspar does not come with any functions for computing the inverse of a matrix, thus this had to be implemented. The chosen method was to augment the matrix A to invert with the identity matrix I, then apply Gauss-Jordan elimination [25] using elementary row operations on the matrix until it reached reduced row echelon form. Reduced row echelon form means that every leading coefficient is 1 and is the only nonzero entry in its column [26].

In MATLAB, the functions `mrdivide` (`/`) and `mldivide` (\) can be used for computing matrix inverse. `B/A` is basically $B * A^{-1}$ if A is a square matrix. According the MATLAB user guide, the functions `mrdivide` and `mldivide` [27] (which are used in the MATLAB implementations) use Gaussian elimination to solve the equation $AX = B$, if A is a square matrix. It seemed reasonable to take this general approach also in Feldspar.

Unfortunately, this required using the `forLoop` function with a matrix in its state, which is not what you want to do in Feldspar (see section 4.5.2). It was necessary because Gaussian elimination iterates over the matrix, updating it in each iteration and the calculations depend on the calculations made in a previous state. Perhaps it would have been possible to take another approach, but it was decided to try this anyway and not dig deeper into other algorithms for matrix inversion, mainly because of time concerns.

Besides matrix inversion, *MMSE EQ1* and *MMSE EQ2* were easily implemented using Feldspar's vector and matrix libraries. The only other function which was missing, was a function for complex conjugate transpose, but since both complex conjugate and transpose were supplied, this was easy.

For both versions, additional implementations were made using core arrays (*MMSE EQ1 core* and *MMSE EQ2 core*) and matrices rather than the vector and matrix libraries (see section 2.1.3). This was done because it was suspected that core arrays and matrices might yield slightly better performance, but also less readable code. Optimized versions using input vectors of explicit lengths, as described in section 2.1.5, were also made for *MMSE EQ1* and its core version (*MMSE EQ1 wrap* and *MMSE EQ1 core wrap*).

MATLAB Implementation

Implementations of both *MMSE EQ1* and *MMSE EQ2* were supplied by Ericsson. Both implementations only needed minor adaptions in order to be compatible with MATLAB Coder; it was sufficient to explicitly tell MATLAB that the input matrices should be complex.

Both implementations were sent to MathWorks for feedback, and they had some minor comments. For *MMSE EQ1*, the code was a bit rearranged, resulting in better readability and possibly slightly better C code. For *MMSE EQ2*, the same thing was done and the calculation of the identity matrix was removed. Instead of first forming the full identity matrix (using `eye`) and then add it, a 1 was added to only those elements which would be affected by an addition with the identity matrix. The original versions (*MMSE EQ1* and *MMSE EQ2*) as well as the optimized versions (*MMSE EQ1 opt* and *MMSE EQ2 opt*) were kept.

3.4 C Code Generation

3.4.1 Feldspar

To compile the Feldspar test programs into C code, the `compile` function was used as described in section 2.1.5 A custom made set of compiler options, `ansiOpts`, was used as described in section 4.5.1. Every function was compiled into a separate C file, containing only one C function with everything inlined.

It was decided to not explicitly set the lengths of the input vectors using the method *wrap* desccribed in section 2.1.5 as default. The reason is that the method was not very well documented, and its importance was not very clear. Versions of some test programs were still made to evaluate the method slightly. These versions are named *wrap*.

3.4.2 MATLAB

The MATLAB test programs were compiled using the `codegen` command (described in section 2.2.3 and using the compiler options described in appendix A.5. Function inlining was always used so that every function compiled into one C function in one file.

As described in section 2.2.3, there are three settings for memory allocation. The method used for all test programs in this thesis is static allocation with hard-coded sizes in the generated C code.

The reason for this is that the other method for static allocation generated some overhead code for checking index bounds etc. Because of this, three files (one for each size) had to be generated.

As discussed in section 5.3.1, MATLAB's weak type system lets the programmer supply arguments of any type to a function. In C however, the types need to be specified, which makes it necessary to state the input and output types when compiling MATLAB functions. This is done by supplying an input example when compiling (this is described in section 2.2.3).

When compiling the MATLAB test programs, it was noted that floating-point variables in MATLAB became variables of type `double` in the C code. In the C code generated by Feldspar, the type `float` was used. This would make the evaluation unfair. MATLAB uses the type `real_T` for floating-point variables, which is defined as `double` in the generated file `rtwtypes.h`. The file was changed to instead define `real_T` as `float` to solve the problem.

Chapter 4

Hard Measures (Performance)

4.1 Method

The method used for the hard measures was to run the compiled C code, generated from the test programs in chapter 3, on two different platforms. The reason for running the code on two platforms was to make the performance evaluation more platform independent, which was an objective of this thesis (see section 1.5). Execution time, memory consumption and number of lines in the generated code was taken into account. The test programs were run with three different input sizes in order see how execution time varied with input size. The number of lines in the generated code might not be relevant at all to the performance, but if the difference is huge between the two languages, it might still be an interesting observation.

To measure execution times, a program was written in C, which reads the input data and then calls all the generated functions in sequence.

The first platform used was a HP EliteBook 8440p, with an Intel Core i5 CPU at 2.4 GHz and 2 GB RAM running 32-bit Windows Vista Enterprise Service Pack 1. This computer was supplied by Ericsson, and it seemed easiest to use this for the benchmarks. Microsoft Visual Studio 2010 Ultimate was used to compile the C code and also to profile the speed.

The second platform was a C6670 simulator from Texas Instruments. "The TMS320C6670 Multi-core Fixed and Floating Point System on Chip is a member of the C66xx SoC family based on TI's new KeyStone Multicore SoC Architecture designed specifically for high performance applications such as software defined radio, emerging broadband and other communications segments" [28]. This was recommended by Ericsson, since it is a new and interesting architecture. Also, it has floating point support, which was needed in this thesis. Texas Instruments Code Composer Studio 4 was used to compile the C code, run the simulator and for profiling.

The memory consumption was calculated by hand. Since no code used any dynamic memory allocation, this seemed easiest. Only arrays were considered for the memory consumption, since the large size of the input vectors made memory for variables negligible.

4.2 PC Benchmark

The project settings used in Visual Studio were default, except for:

- The code was set to be compiled as C code.
- The optimization level was set to "Full Optimization".
- For the small test programs, "Inline Function Expansion" was disabled because some of them would otherwise not be seen in the profiling results.

The profiling method used for the PC benchmarks was sampling of CPU clock cycles, which was the recommended method in Visual Studio because of high accuracy. The sampling frequency was set to 1 sample per 150000 clock cycles, which was the highest frequency available. High sampling frequency was desired since it would yield high accuracy, and since small functions otherwise would not be sampled at all, as discussed in the section below. Inclusive samples were counted, meaning that samples in functions called by the test functions were included.

Because of the relatively large sample intervals in the PC profiling, small functions are sometimes not sampled at all, meaning that they do not appear in the results. The solution to this was to simply loop the function calls to such small functions. Then a new possible problem arose. Sampling could for instance always occur at the for-loop conditional and not inside the function. However since most test programs loop at least to the length of a rather long input vector, the vast majority of clock cycles in a looped function call will actually belong to the function's body, making such sampling misses negligible. To be able to compare test programs between Feldspar and MATLAB, small functions were of course looped the same number of times for both languages.

4.3 TI C6670 Simulator Benchmark

The projects settings used in Code Composer Studio were default, except for:

- The optimization level was set to 3.
- "Optimize for speed" was set to 5.

The profiler in Code Composer Studio counts all clock cycles of the simulations, as opposed to the sampling used in the PC benchmarks.

4.4 Input Data

For all test programs, three different runs with different size of input data was made. The idea was to see how the execution time varied with input size. The input sizes for the TI simulator test runs had to be much smaller than those for the PC test runs, because the simulator ran much slower.

The input data for the small test programs (see section 3.2) was generated in MATLAB and then written to files, which were read by the main test program. For all functions except *TwoFir* and

BinarySearch, the input data was random numbers. For *BinarySearch* a random but sorted list was generated, and for *TwoFir* a nice sound of a train honk was used. The train honk sound was obtained in MATLAB by the command `load train`. The `fir1` function in MATLAB was finally used to generate the filter coefficients for *TwoFir*.

For the DSP test programs *ChEst* (section 3.3.3) and *MMSE EQ* (section 3.3.4), the input data was obtained from simulation files which were provided by Ericsson together with the MATLAB implementations of the test programs. The input data of one simulation run was written to files, which could later be read by the main test program.

For *ChEst*, the parameter changing between the test runs was the number of sub-carriers. This decided the output size, which seemed important for the performance.

For *MMSE EQ*, the variable between the test runs was the size of the input matrices. Since the matrices are calculated independently of each other (independent of which sub-carrier), the number of sub-carriers was not changed between the three runs.

For *DRS* (section 3.3.2), no input data was necessary to generate. The only parameter that really affected the performance was the number of sub-carriers, since it decides the size of the output vector. The number of sub-carriers was thus the only parameter changed between the test runs.

The input data used when running the test programs is found in the tables below:

Test Program	PC Run 1	PC Run 2	PC Run 3	TI Run 1	TI Run 2	TI Run 3
BinarySearch	25000	50000	100000	100	200	400
BitRev	16384	32768	65536	16	32	64
Bubblesort1	250	500	1000	10	20	40
Bubblesort2	250	500	1000	10	20	40
Bubblesort opt	250	500	1000	10	20	40
DotRev	25000	50000	100000	100	200	400
DotRev opt	250	500	1000	10	20	40
Duplicate	25000	50000	100000	100	200	400
Duplicate2	25000	50000	100000	100	200	400
RevRev	25000	50000	100000	100	200	400
SqAvg	25000	50000	100000	100	200	400
SqAvg opt	25000	50000	100000	100	200	400
TwoFir	3220	6440	12880	128	256	512
TwoFir wrap	3220	6440	12880	128	256	512
ChEst	2048	2048	2048	64	64	64
ChEst wrap	2048	2048	2048	64	64	64
MMSE EQ1	$1200*2*2$	$1200*4*4$	$1200*8*8$	$12*2*2$	$12*4*4$	$12*8*8$
MMSE EQ1 core	$1200*2*2$	$1200*4*4$	$1200*8*8$	$12*2*2$	$12*4*4$	$12*8*8$
MMSE EQ1 wrap	$1200*2*2$	$1200*4*4$	$1200*8*8$	$12*2*2$	$12*4*4$	$12*8*8$
MMSE EQ1 core wrap	$1200*2*2$	$1200*4*4$	$1200*8*8$	$12*2*2$	$12*4*4$	$12*8*8$
MMSE EQ1 opt	$1200*2*2$	$1200*4*4$	$1200*8*8$	$12*2*2$	$12*4*4$	$12*8*8$
MMSE EQ2	$1200*2*1$	$1200*4*2$	$1200*8*2$	$12*2*2$	$12*4*2$	$12*8*2$
MMSE EQ2 core	$1200*2*1$	$1200*4*2$	$1200*8*2$	$12*2*2$	$12*4*2$	$12*8*2$
MMSE EQ2 opt	$1200*2*1$	$1200*4*2$	$1200*8*2$	$12*2*2$	$12*4*2$	$12*8*2$

Table 4.1: Number of input elements for each test program. Test programs which don't take any input data are not shown.

Test Program	PC Run 1	PC Run 2	PC Run 3	TI Run 1	TI Run 2	TI Run 3
IdMatrix	$250 * 250$	$500 * 500$	$1000 * 1000$	$10 * 10$	$20 * 20$	$40 * 40$
SliceMatrix	$125 * 175$	$250 * 350$	$500 * 700$	$5 * 7$	$10 * 14$	$20 * 28$
ChEst	$N_{Sc} : 600$	$N_{Sc} : 900$	$N_{Sc} : 1200$	$N_{Sc} : 12$	$N_{Sc} : 24$	$N_{Sc} : 36$
ChEst wrap	$N_{Sc} : 600$	$N_{Sc} : 900$	$N_{Sc} : 1200$	$N_{Sc} : 12$	$N_{Sc} : 24$	$N_{Sc} : 36$
DRS	$N_{Sc} : 600$	$N_{Sc} : 900$	$N_{Sc} : 12000$	$N_{Sc} : 36$	$N_{Sc} : 48$	$N_{Sc} : 60$
DRS opt	$N_{Sc} : 600$	$N_{Sc} : 900$	$N_{Sc} : 12000$	$N_{Sc} : 36$	$N_{Sc} : 48$	$N_{Sc} : 60$

Table 4.2: Varying input arguments for test programs with parameters. For $IdMatrix$ and $SliceMatrix$, the number of elements in the output matrix is shown. For SliceMatrix, the matrix to slice was an identity matrix of 40×40 elements for TI, and 1000×1000 for PC.

4.5 General Problems

4.5.1 Complex Numbers

The Feldspar compiler generates C code according to the ISO C99 standard. Since neither of the C compilers used in this thesis supported ISO C99 to its full extent, this became a problem. The generated C99 code was compliant with the compilers for all test programs except for those where complex numbers were used. The ISO C99 standard uses a type qualifier `complex` and some basic functions for complex arithmetic which was not supported by the compilers.

The Feldspar compiler offers in-depth control of how code is generated by letting the user provide compiler options (described in section 2.1.5). There are predefined sets of compiler options and they may be modified to suit the user's needs. Modifying the compiler options was the key to solving the problem with complex numbers. The compiler option `defaultOptions` was modified to not use the `float complex` type and instead use `complexOfFloat`, which is used in the compiler options `tic64xPlatformOptions`. The new compiler options module was named `ansiOpts` to denote that it was compliant with the older ANSI C standard (at least for the test programs used in this thesis).

Arithmetic functions in Feldspar generate different function calls in the C code, depending on the platform options described above. The corresponding C functions are supplied in separate C files which come with Feldspar. The files are `feldspar_c99.h` and `feldspar_tic64.h`, and they contain the C functions for arithmetic for the respective platform. Because the custom made compiler options `ansiOpts` is a mix of the C99 and tic64x options, a mix of the C files had to be created as well. This is called `feldspar_ansi.h` and is the same as `feldspar_c99.h` except that all functions concerning complex arithmetic on floats have been replaced by their corresponding functions from `tic64.h`.

The reason for not using `tic64xPlatformOptions` directly is that it perhaps would generate code with platform specific intrinsics, which would not match the platforms used in this thesis.

4.5.2 Vector as State in the forLoop Function

One big problem, which proved to be the most severe problem with Feldspar in this thesis, arises when one needs to have a vector as state in the forLoop function. This is a fundamental problem which derives from Feldspar's property of being referentially transparent. If a function takes a vector and returns a vector, ie. has the type `Vector a -> Vector a`, the input vector must remain unchanged after an application of the function (if the input is used later). This is because the function must always yield the same result on the same input in order to be referentially

transparent. If the function updates the input vector in-place, it would mean that a second application of the function might not at all yield the same result.

The result of this, is that a copy of the input vector must be made, and has to be copied in each iteration of the for-loop. Consider the following function taking a core array of length `10`, which for each iteration, adds `1` to the element at the current index:

```
add1 :: Data [Int32] -> Data [Int32]
add1 v = forLoop 10 v (\i st -> setIx st i (st!i + 1))
```

This compiles to the following C code:

```
void test(struct array mem, struct array in0, struct array * out1)
{
    copyArray(out1, in0);
    {

     uint32_t i3;

     for(i3 = 0; i3 < 10; i3 += 1)

     {

         copyArray(&at(struct array,mem,0), (* out1));

         at(int32_t,at(struct array,mem,0),i3) = (at(int32_t,(* out1),i3) + 1

);

         copyArray(out1, at(struct array,mem,0));

     }
    }
}
```

As we can see, the input array is never changed. The problem with this is of course the copies made by `copyArray`, which results in both higher memory consumption as well as slower execution. See section 2.1.5 for more details about memory and a description of `struct array` and `at`.

In many cases, this way of programming can be avoided by, for example, using the vector library as described in section 2.1.3. However, there are some cases where it cannot be avoided. This is usually when a computation of an element in a vector is dependent on a previous computation made inside a conditional. See the *BubbleSort* test program in section 3.2 for an example of this.

4.5.3 Memory

In Feldspar, it is up to the programmer to allocate memory for all arrays used in the C code. This memory is provided as an argument to the function as a special nested `struct array` called `mem` (see section 2.1.5). To construct `mem`, it is currently necessary to manually look through the generated C code and figure out the structure of `mem`. For larger examples, this is not a trivial task.

For example, the *MMSE EQ* test program (see section 3.3.4) needed almost 40 array structures in `mem`, and it was a mixture of both one dimensional and two dimensional arrays of different lengths, which made it very tedious to write the C code for setting up `mem` prior to call the actual function. This actually resulted in that one of the test programs (*MMSE EQ2* using the vector library) did not work correctly, probably because of `mem` being erroneously set up. This implementation of *MMSE EQ2* in Feldspar was therefore not included in the test runs.

In MATLAB, all memory needed by the functions was allocated on the stack. Since MATLAB does not suffer from the problem with unnecessary array copies (see section 4.5.2), the number of arrays are low, meaning that the stack memory used will not grow out of control.

4.6 Results: Execution Time

In this section, a selection of the results are presented and explained. All results are presented in appendix B. Many results were very similar, especially between PC and TI, which is why not all results are presented here. The result graphs in the sections below show the number of clock cycles acquired from the profiling versus the number of input elements.

In the result graphs of the test programs *TwoFir*, *ChEst*, and *MMSE EQ* below, single points can be found labeled *feldspar wrap*. These are the execution times of versions of the test programs where the lengths of the input vectors were explicitly set, as discussed in section 2.1.5. The reason for only testing these versions with one input size is that it was considered enough to compare one point to see how the performance was affected. The results are however interesting since the execution times are sometimes much lower than for the regular Feldspar implementations.

4.6.1 BubbleSort

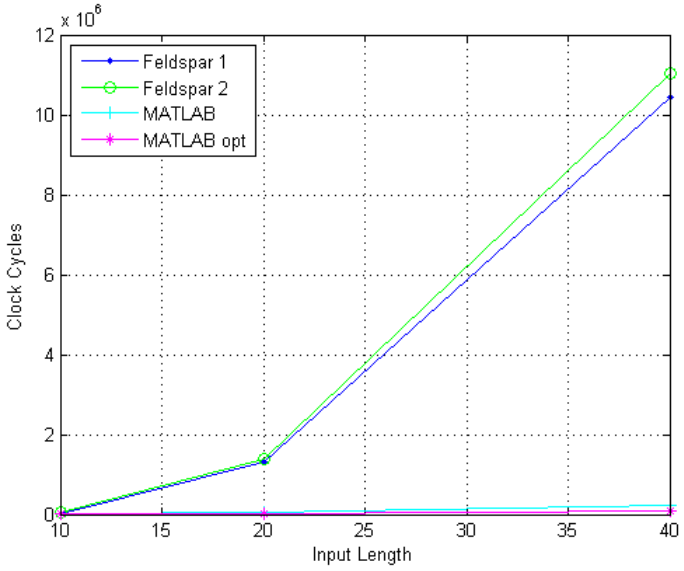

Figure 4.1: *BubbleSort* run on TI simulator with input vector sizes 10, 20 and 40.

The Feldspar implementations of *BubbleSort* generate C code containing a lot of array copying because they both use the a vector as state in the `forLoop` function (see section 4.5.2). The

generated C code from the MATLAB implementations updates in-place which is why they run so much faster.

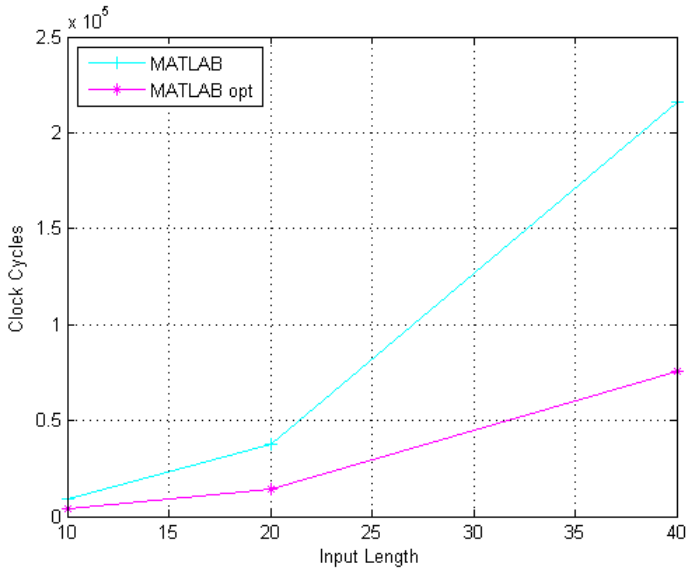

Figure 4.2: Figure 4.1 zoomed in on the MATLAB implementation.

Figure 4.2 shows that the MATLAB implementation grows in the same way as the Feldspar implementation, but with a much smaller constant factor.

4.6.2 DotRev

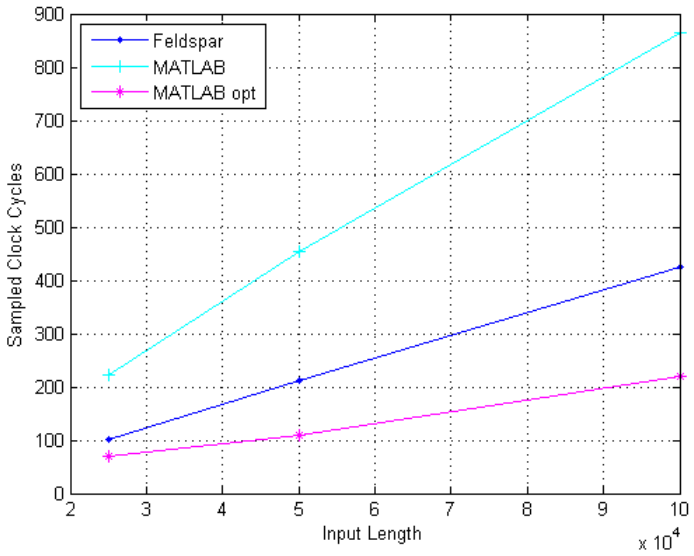

Figure 4.3: *DotRev* run on PC with input vector size of 25000, 50000 and 100000 elements.

Fusion seems to give Feldspar an advantage compared to the non-optimized version in MATLAB. However, the optimized version in MATLAB performs better than the Feldspar version. The reason for this is the slightly more static code of MATLAB (hard-coded sizes).

4.6.3 SliceMatrix

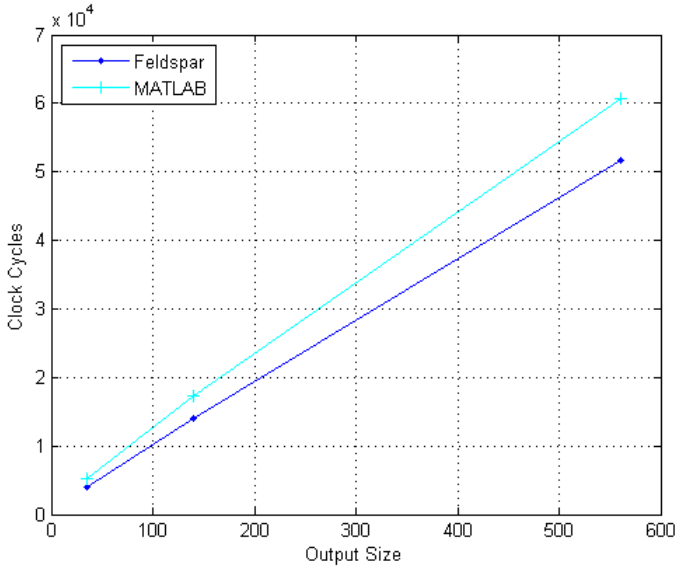

Figure 4.4: *SliceMatrix* run on TI computing a slice of size 5×7, 10×14 and 20×28 of a 40×40 identity matrix.

The Feldspar implementation only computes the elements inside the slice, while the MATLAB implementation first computes the full matrix and then the slice. This is due to fusion, and is why Feldspar performs better in this case.

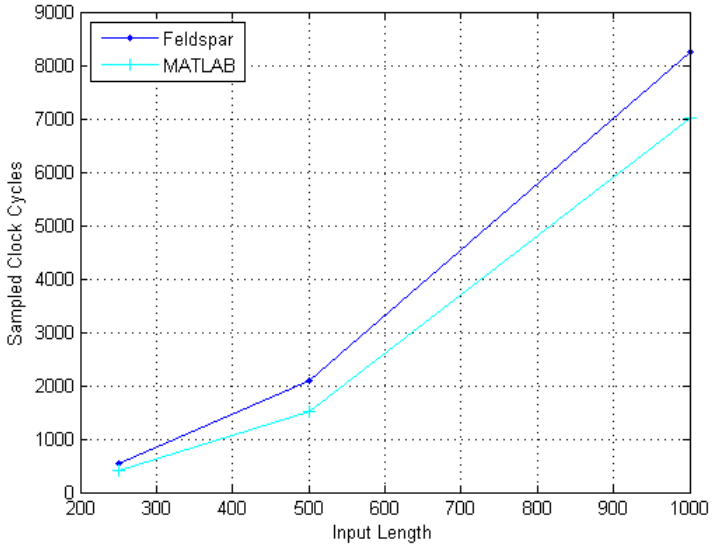

Figure 4.5: *SliceMatrix* run on PC computing a slice of size $125 * 175$, $250 * 350$ and $500 * 700$ of a 1000×1000 identity matrix.

In the PC run, MATLAB performs better than Feldspar. The reason is probably that the generated code from MATLAB uses hard-coded sizes, or that the computation of the identity matrix is faster on the PC.

4.6.4 SqAvg

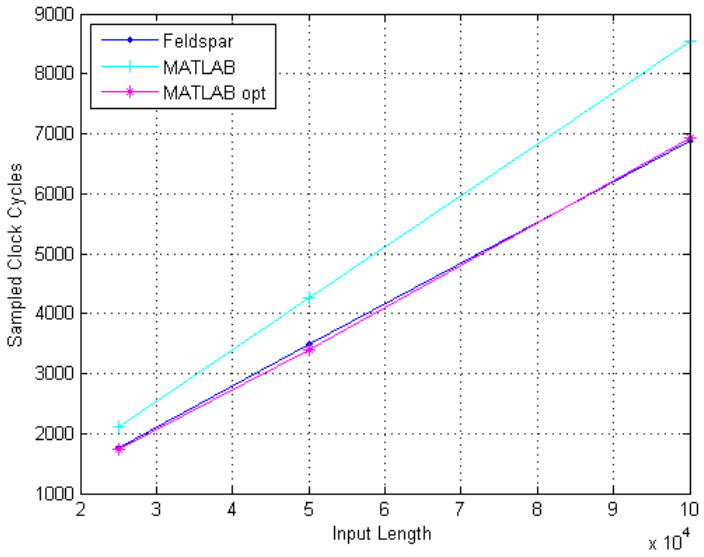

Figure 4.6: *SqAvg* run on PC with input vector size of 25000, 50000 and 100000 elements.

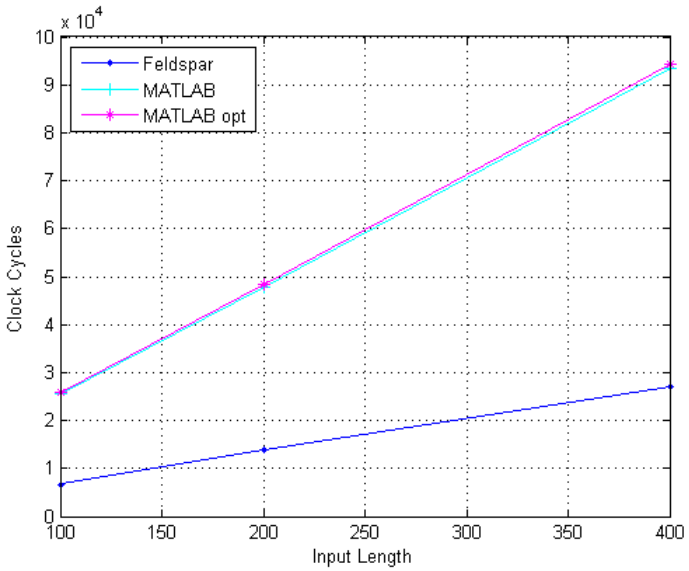

Figure 4.7: *SqAvg* run on TI simulator with input vector sizes 100, 200 and 400.

The difference between the results in figure 4.6 and in figure 4.7 is that the optimized and non-optimized MATLAB implementations in figure 4.7 yielded the same results. This is probably because of the unnecessary counters introduced in the optimized MATLAB implementation (described in section 5.3.3) and differences between the compilers/platforms.

4.6.5 TwoFir

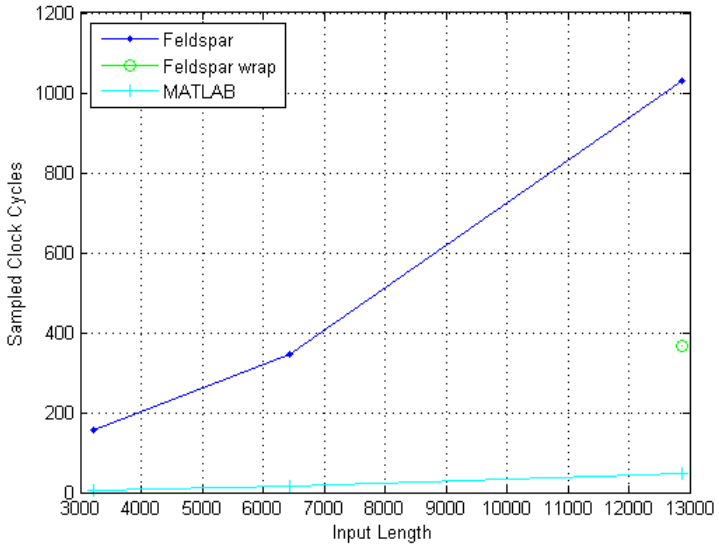

Figure 4.8: *TwoFir* run on PC with input vector size of 3220, 6440 and 12880 elements, and filter coefficients of 5, 10 and 20 elements.

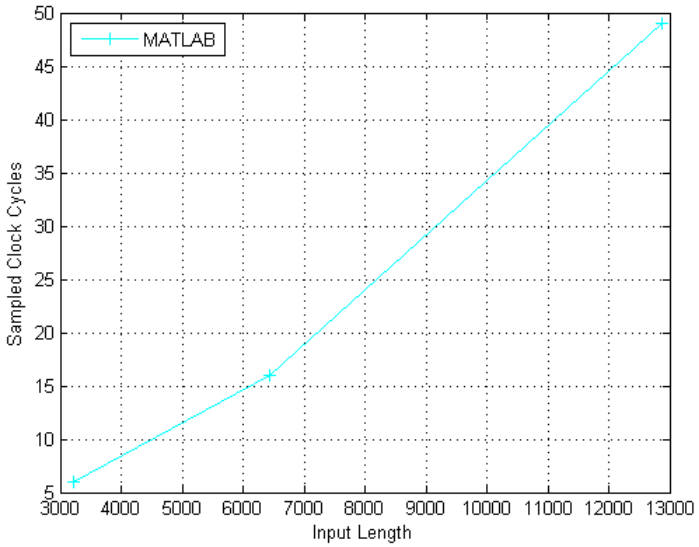

Figure 4.9: Figure 4.8 zoomed in showing only MATLAB.

In figure 4.8, the MATLAB implementation is clearly much faster than the Feldspar implementation. However, the Feldspar implementation using input vectors of explicit lengths (*wrap*) and no fusion shows a big difference in performance. This implementation stores the result of the first filter and does not have to make repeated computations (see section 2.1.4). Also, wrap avoids some unnecessary copies, which otherwise affects the performance negatively.

The scaling makes the MATLAB implementation look linear in the first figure, which is why a zoomed version is added as the second figure (4.9). This shows that the growth is similar to the Feldspar implementation.

4.6.6 AddSub, TransTrans and RevRev

For *AddSub* and *TransTrans*, MATLAB generated only an empty function, which should be the desired behaviour. For *RevRev*, two for-loops were generated reversing the input vector twice. The difference between *AddSub* and *Transtrans* and *RevRev* is that for the first two, matrix operations are used. As mentioned in section 5.3.2, this usually generates better code than anything else in MATLAB, which is probably why *AddSub* and *TransTrans* generated better code than *RevRev*, which used the function `fliplr`.

In Feldspar, these test programs generated code where the multiple applications were fused into one for-loop. The compiler was not able to completely remove the computations, as in MATLAB.

```
void addSub(struct array mem, struct array in0, struct array * out1)
{
    setLength(out1, length(in0));
    {
        uint32_t i2;
        for(i2 = 0; i2 < length(in0); i2 += 1)
        {
            at(int32_t,(* out1),i2) = ((at(int32_t,in0,i2) + 1) - 1);
        }
    }
}
```

C code generated from *AddSub* in Feldspar. The addition and subtraction is fused into one for-loop, but not removed completely as in MATLAB. For *TransTrans*, a for-loop where each row in the input is copied to the output is generated:

```
void transTrans(struct array mem, struct array in0, struct array * out1)
{
    uint32_t v2;

    v2 = length(at(struct array,in0,0));
    setLength(out1, length(in0));
    {
        uint32_t i3;
        for(i3 = 0; i3 < length(in0); i3 += 1)
        {
            copyArray(&at(struct array,(* out1),i3), at(struct array,in0,i3));
            setLength(&at(struct array,(* out1),i3), v2);
        }
    }
}
```

4.6.7 Demodulation Reference Symbols (DRS)

There is a clear gap in performance between Feldspar and MATLAB for the *DRS* test program. The different implementations are described in section 3.3. There are only results from the optimized Feldspar implementation, since the non-optimized resulted in too long execution times.

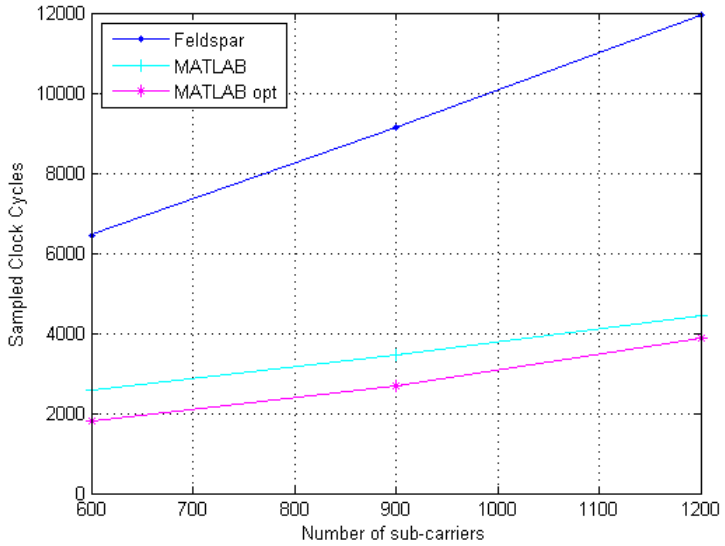

Figure 4.10: *DRS* run on PC with 600, 900 and 1200 sub-carriers.

The DRS algorithm uses a vector containing the first prime number below each multiple of 12. The C code generated by the Feldspar implementation constructs this vector every time it is used (even inside loops), and not just once which had been preferred. If it had been defined just once, chances are that the Feldspar implementation had been on par with the MATLAB implementations or even better, since the Feldspar code would have been significantly shorter than any of the MATLAB implementations generated into C code (see section 4.8).

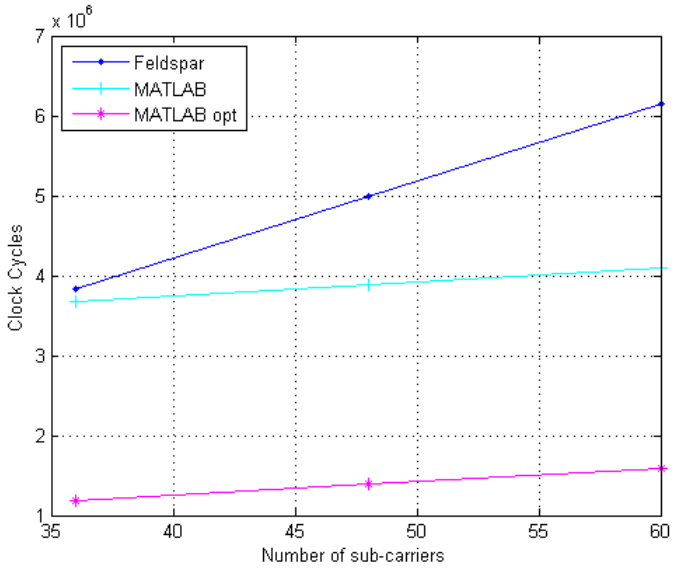

Figure 4.11: *DRS* run on TI simulator with 36, 48 and 60 sub-carriers.

The non-optimized MATLAB implementation is much closer to the optimized when run on PC as shown in figure 4.10, than in the TI simulator run shown in figure 4.11. Possible reasons are the different input values and the fact that the C compilers might have optimized the C code differently.

4.6.8 Channel Estimation

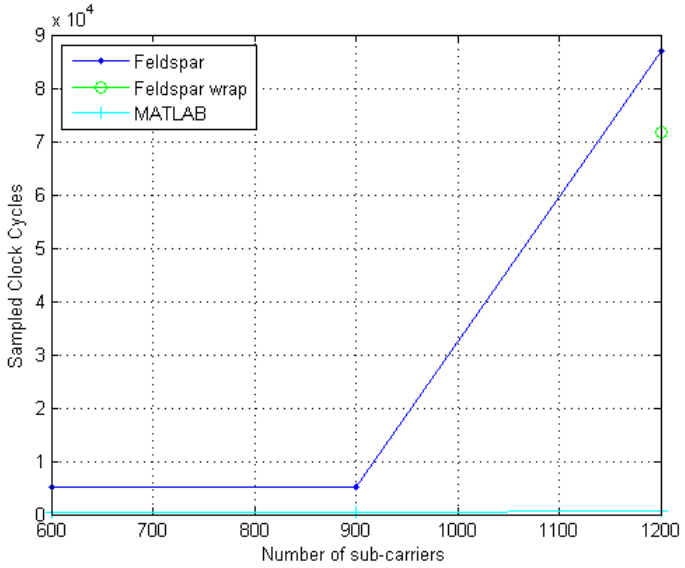

Figure 4.12: *ChEst* run on PC with 600, 900 and 1200 sub-carriers.

The MATLAB implementation is clearly much faster than the Feldspar implementation. MATLAB's FFT and IFFT are probably better, as discussed in section 3.3.3. Also, the unnecessary copies generated from the Feldspar code have bad impact on the performance. The Feldspar version using input vectors of explicit lenghts (wrap) slightly improved the performance, probably because of fewer lines of code (see section 4.8).

Since the output size of this function is the nearest above power of two, 600 and 900 was not a very good choice for the first two input sizes. The output size for the two first runs are both 1024, which is why they did almost not differ in sampled clock cycles.

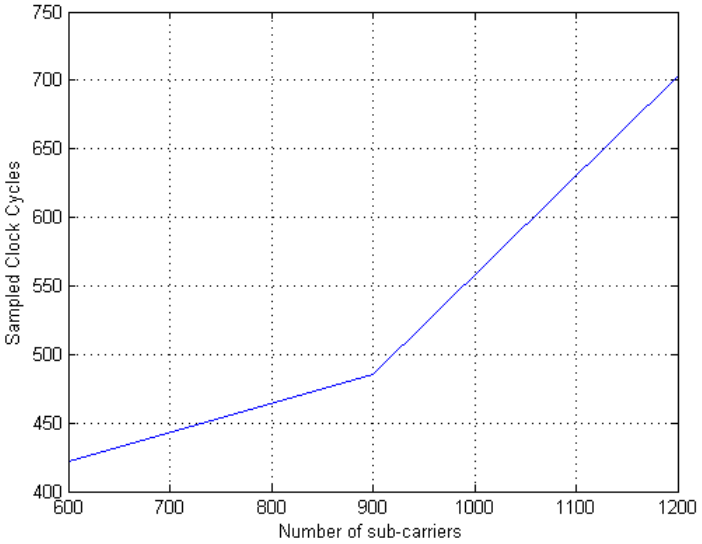

Figure 4.13: Figure 4.12 zoomed in showing only MATLAB.

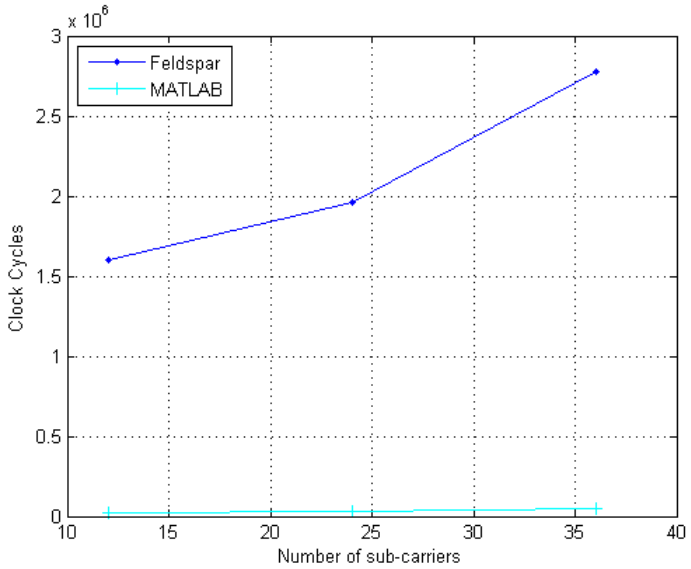

Figure 4.14: *ChEst* run on TI simulator with 12, 24 and 36 sub-carriers.

When run on the TI C6670 simulator, the output sizes to the respective inputs were 32, 64 and 128, which explains why the Feldspar curve grows smoother than in figure 4.12.

4.6.9 Minimum Mean Square Error Equalization (MMSE EQ1)

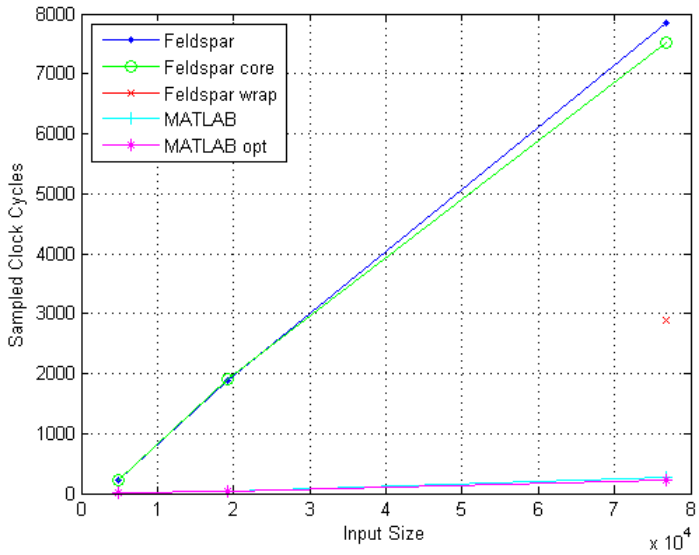

Figure 4.15: *MMSE EQ1* run on PC with matrix size 2×2, 4×4 and 8×8 and 1200 sub-carriers. Input Size thus denotes the total number of elements.

The MATLAB implementations shows about 30 times better performance than Feldspar and Feldspar core. The big difference is explained by the matrix inverse used in the Feldspar implementations. MATLAB's computation of the matrix inverse is of course much better than the implementation made for this thesis, which is very simple. The large amount of unnecessary copies of matrices and vectors also contributes to the bad results. However, the Feldspar version using

wrap shows a huge improvement. The number of unnecessary intermediate arrays were cut down from about 40 to 10, and the number of lines of C code from about 1800 to 400.

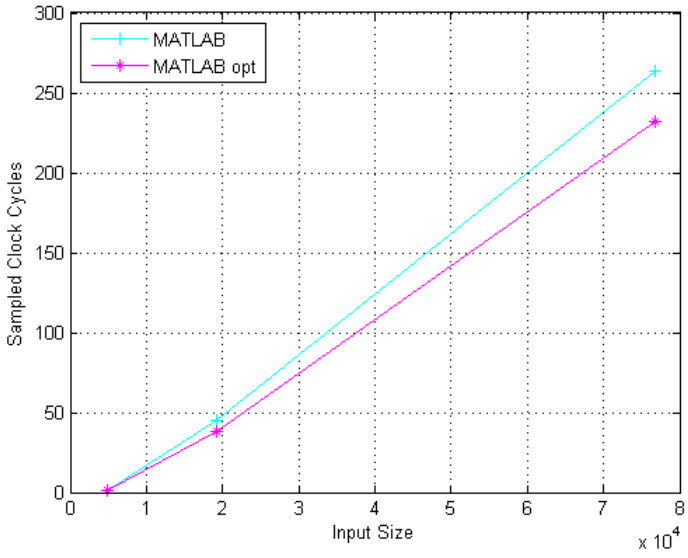

Figure 4.16: Figure 4.15 zoomed in, showing only MATLAB and MATLAB opt.

Figure 4.16 is added to show that the MATLAB implementations grow similar as the Feldspar implementations, and the difference is a constant factor. This factor is rather big, about 30 for Feldspar and Feldspar core, and 10 for Feldspar wrap.

4.7 Results: Memory Consumption

The memory consumption was calculated by hand. Because of the large input data sizes, only arrays were considered and scalar variables were considered negligible. The memory for the input and output was not included, because this is of course the same for both languages. In the tables below, the memory usage is expressed in terms of the size of input and output data and does not consider that different types have different size.

Test Program	Feldspar	MATLAB
AddSub	0	0
BinarySearch	0	0
BitRev	V_{in}	$2V_{in}$
Bubblesort1	$2V_{in}$	0
Bubblesort2	$2V_{in}$	-
Bubblesort opt	-	0
DotRev	0	V_{in}
DotRev opt	-	0
Duplicate	0	0
Duplicate2	V_{in}	-
IdMatrix	0	0
SliceMatrix	0	M
RevRev	0	0
SqAvg	0	V_{in}
SqAvg opt	-	0
TransTrans	0	0
TwoFir	$10K$	V_{in}
TwoFir wrap	V_{in}	-

Table 4.3: Memory consumption for the small test programs. V_{in} is the number of input vector elements, M is the number of matrix elements and K is the number of elements in the coefficient vector. When there is a -, it means that the test program was not implemented in this language.

As seen in table 4.7, *DotRev*, *BitRev*, *SliceMatrix*, and *SqAvg* use more memory in MATLAB than in Feldspar. Because of fusion, there are no intermediate vectors in the C code generated by Feldspar in these programs. Since no intermediate result has to be stored, the final result can be written directly to the output array in the Feldspar test programs.

For *BubbleSort*, the code generated by Feldspar uses more memory because it contains a for-loop with a vector in its state. This results in unnecessary array copying, as explained in section 4.5.2. In MATLAB, the vector updating occurs in-place and does not introduce any copies of the array in the C code.

For *TwoFir*, the non-wrapped Feldspar case has the smallest memory consumption, because of fusion, no copy of the input signal vector is made. However it uses 10 copies of the coefficient vectors which should not be necessary. The wrapped version is implemented to not use fusion because it resulted in faster execution time in this case. Because of the lack of fusion, intermediate arrays are used resulting in higher memory consumption.

As seen in table 4.7, the MATLAB implementation of *ChEst* uses less memory, probably because no unnecessary copies have to be made. However, the MATLAB implementation uses lookup tables for the FFT computations, which introduce a constant factor of memory consumption. These tables are however small compared to the input size, and are thus negligible. The function computing the FFT in Feldspar introduces some unnecessary copies, which is the reason for the higher memory consumption.

Test Program	Feldspar	MATLAB
ChEst	$8V_{in} + 3V_{out}$	$V_{in} + V_{out} + 68$
ChEst wrap	$4V_{in} + 3V_{out}$	-
DRS	8500	13547
DRS opt	316	5522
MMSE EQ1	$16N_{Rx}N_{Tx} + 16N_{Rx}$	$3N_{Rx}N_{Tx} + N_{Rx}$
MMSE EQ1 core	$32N_{Rx}N_{Tx} + 8N_{Rx}$	-
MMSE EQ1 wrap	$7N_{Rx}N_{Tx} + 2N_{Rx}$	-
MMSE EQ1 core wrap	$8N_{Rx}N_{Tx} + 2N_{Rx}$	-
MMSE EQ1 opt	-	$2N_{Rx}N_{Tx} + N_{Rx}$
MMSE EQ2	$16N_{Rx}N_{Tx} + 16N_{Rx}$	$T_{Rx}^2 + N_{Rx}N_{Tx} + N_{Rx} + N_{Tx}^2$
MMSE EQ2 core	$32N_{Rx}N_{Tx} + 8N_{Rx}$	-
MMSE EQ2 opt	-	$T_{Rx}^2 + 3N_{Rx}N_{Tx} + T_{Rx} + N_{Tx}^2$

Table 4.4: Memory consumption for the DSP test programs. V_{in} is the number of input vector elements, V_{out} is the number of output vector elements, N_{Rx} is number of receiving antennas and N_{Tx} number of MIMO streams. When there is a -, it means that the test program was not implemented in this language.

DRS uses more memory for both MATLAB implementations. In the optimized Feldspar version, fusion results in no intermediate vectors which means that the computations from the input to the output can be done in one step. However, three copies of the same prime table is stored in different arrays, which should not be necessary. These copies only introduce a constant factor of memory consumption, which is relatively small compared to the total amount of memory. In the naive Feldspar implementation, unnecessary copies of the vector containing the bit sequence were introduced, resulting in higher memory consumption.

In the *MMSE EQ* test programs, the Feldspar implementations not using wrap introduced a large amount of unnecessary copies of matrices and vectors. This resulted in very high memory consumption compared to the MATLAB implementations, which use in-place matrix/vector updating. However, the Feldspar implementations using wrap resulted in significantly less unnecessary copies.

4.8 Results: Lines of Generated Code

The generated C code of the DRS implementations in MATLAB had a little more lines of code than the C code from the Feldspar implementations, which can be explained by Feldspar's fusion. However, the Feldspar implementations resulted in three occurrences of a large table of prime numbers when only one would have been necessary. If this prime table had only occurred once, there would have been much fewer lines.

The *MMSE EQ* implementations in Feldspar not using wrap, contains many unnecessary copies of arrays and also a lot of strange if-clauses. The implementations using wrap helped the compiler a bit, resulting in much less code, almost equal to the number of lines of the MATLAB implementation.

Test Program	Feldspar	MATLAB
AddSub	11	0
BinarySearch	50	33
BitRev	42	35
Bubblesort1	38	28
Bubblesort2	59	-
Bubblesort opt	-	23
DotRev	18	23
DotRev opt	-	12
Duplicate	22	5
Duplicate2	24	-
IdMatrix	18	9
SliceMatrix	64	43
RevRev	16	20
SqAvg	18	17
SqAvg opt	-	18
TransTrans	15	0
TwoFir	109	42
TwoFir wrap	79	-
ChEst	302	236
ChEst wrap	159	-
DRS	333	357
DRS opt	411	222
MMSE EQ1	1833	290
MMSE EQ1 core	1259	-
MMSE EQ1 wrap	375	-
MMSE EQ1 core wrap	309	-
MMSE EQ1 opt	-	282
MMSE EQ2	1715	513
MMSE EQ2 core	1259	-
MMSE EQ2 opt	-	499

Table 4.5: Number of lines in the generated C code for each test program. When there is a $-$, it means that the test program was not implemented in this language.

Chapter 5

Soft Measures (Productivity)

5.1 Method

This section describes the method used for evaluating the productivity of Feldspar and MATLAB. Since productivity is something highly subjective, the method of an evaluation like this is not trivial.

Interesting areas where scientific studies have been done were selected. The areas are defined in section 5.2, sometimes together with criteria describing what needs to be fulfilled by a language in order to be considered good in the area. The areas were chosen as described in section 1.6. The criteria were chosen based on scientific studies and the personal opinions of the authors of this thesis.

The languages were evaluated by reasoning about them in each area, sometimes using examples from the test programs in chapter 3. The evaluation can be found in section 5.3, which follows the same structure as the definitions for easy reading.

Also, a small survey was put together to get a wider perspective. It contained questions about reading code and making small changes to it in both languages. The survey is found in appendix C together with the answers. As mentioned in section 1.6, the results of the survey were not used to draw any conclusions, but rather to discuss some interesting indications. The reasons for this are the low number of participants and the non-scientific nature of the survey. See section 5.4 for more information about the survey and a discussion of the results.

5.2 Definitions

5.2.1 Maintainability

Every software system needs to be changed in order to meet requirements from its users. This is called maintenance. The effort spent on maintaining a software system is reported to be around 70% of the total development and support efforts [29]. Maintainability places high demands on both the source code and the documentation. It can be almost impossible to understand code written by someone else if it has poor readability (see section 5.2.3) or documentation. When

changing the functionality, it is also important not to break existing functionality which might rely on code that is changed. Consider a function which is called from two different locations in a program, where the first call never passes a negative parameter while the second one does. It might be the case that the function has to be changed and the developer then forgets to implement the negative case, which may cause devastating errors.

Type Safety

A type system of a programming language associates expressions in the language according to the types of values they compute [30]. The type system tries to prove that no type errors can occur in a program. What is meant by a type error is determined by the type system, but generally it means that an operation was performed on values which are not appropriate for that operation, for example dividing a string with an integer.

A type system can be static, which means that the types are checked at compile-time. A type system can also be dynamic, where most types are checked at run-time. It can also be strongly or weakly typed, which decides to what extent conversions are allowed. A type system with strong typing will prevent an operation with arguments that have the wrong type. For example, adding a string with an integer would be illegal in a strongly typed language, while in a weakly typed language the result of this operation would be unclear. One language might convert the string to a number, while another might convert the integer to a string, a shown in section 5.3.1.

Debugging

When a piece of software does not behave as intended, one probably wants to trace the execution in order to find out what is going wrong. There are several possibilities for this; the simplest is perhaps to manually print out the values which are of interest, but the by far most useful method is to use a debugger. A source-level debugger typically lets you step through the original source code line by line, examining variable content during execution. This should be the preferred method considered in this thesis.

Documentation

When documenting source code, it is important that it is relevant and concise. It is stated that useless comments are worse than missing ones, because they can confuse or sidetrack the developer [31]. Also, the DRY (Don't Repeat Yourself) principle [32] states "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." This really gives motivation for having the documentation inside the source code rather than in a separate document. In this way it is less likely that some parts of the code are updated but the corresponding documentation is not. The documentation for a function should at least contain:

- A brief description of what the function does.
- A description of the functions arguments and return value.

Another advantage of having the documentation and source code in the same document, is that there are various tools for different languages which automatically extract information from the source files. Special comments are often used to add information to the documentation.

5.2.2 Naive vs. Optimized

When programming, it might be the case that the programmer first chooses to solve the problem in the way he finds more intuitive. This will obviously not always be the solution that yields best performance, and the programmer will be required to make optimizations to the code in order to improve it. The final solution might be very different from the original idea; it will probably have better performance, but in many cases the code will also be harder to read. Here it is important to consider if optimizations are wanted. Non-optimized code could actually be preferred over optimized code if performance is not essential but readability is.

```
for (int i = 0; i < n; i++)
{
        a[i] =  a[i] * 2;
}
```

```
for (int i = 0; i < n; i+=4)
{
        a[i] <<= 1;
        a[i+1] <<= 1;
        a[i+2] <<= 1;
        a[i+3] <<= 1;
}
```

Table 5.1: Both examples contain for-loops which multiplies each element in an array by 2. However, the right example uses loop unrolling as well as left shift for multiplication, which might improve performance but make it slightly harder to read.

It would be good if the compiler figured out the clever things instead, relieving the programmer from both the consideration of whether optimizations are necessary and the actual implementation of these optimizations. The programmer then does not need to know specific language/compiler implementation details in order to write code that in turn generates good C code. This means that the gap in performance of the C code generated from optimized and non-optimized implementations should be as small as possible.

5.2.3 Readability

In this thesis, readability will be considered both for the high level code and the generated code. The reason for this, is that even if it would be optimal to just have to concentrate on the high level code and fully rely on the compiler for the generated code, it is probably necessary to sometimes look in the C code for debugging purposes, or to use it as reference code. It is then very important that the generated code is possible for a human to understand.

Most programmers have their own opinions of what readability is and what affects it, making it a very subjective matter. In this thesis, guidelines from a book concerning readability [33] were taken into account.

The results from a research paper with the purpose of finding aspects which are important for readability were also taken into account when evaluating readability [34]. In this paper, machine learning algorithms were used on data from the results of a study about readability.

Identifier Naming

According to [33] , the naming of identifiers is very important to make a software system understandable. The name of variables gives the reader a lot of information about the functionality.

One conclusion by [34] was that the average length of identifier names in a code snippet had nearly no importance at all for the readability. However, the maximum identifier length showed to have negative impact. This means that one should be able to use one-character variable names without sacrificing readability, while being careful not to use too long ones.

Line Length

Another conclusion discussed in [34] is that the number of identifiers and characters per line of code greatly affects the readability. Long lines of code also proved to be much less readable than short ones. [33] agrees on this in the sense that one line should not contain more than one statement or declaration.

Uneccessary Code

Code that is unnecessary, for example unnecessary variables or a conditional which always evaluates to true, will only confuse the reader and make the code harder to understand. Unnecessary code can also mean code which is unnecessarily complicated. This will be considered for the generated code.

Levels of Abstraction

Programming languages are said to be on different levels of abstraction, meaning that they hide different amounts of implementation details from the user. The lower level, the more machine-near the code tends to be, and the higher level, the more abstract. High level code aims to be more user friendly since the programmer does not need to be as much concerned about the machine the program is run on as in the low level case.

Different versions of a program can often be coded at different abstraction levels in the same programming language, meaning that the used language features hide a certain amount of details from the programmer. A good reference to see if a version of a program is at a high or low abstraction level is to compare it to a mathematical specification of the program.

A mathematical specification is defined as a specification using mathematics to express an algorithm. An example of a mathematical specification is found in section 5.3.3.

The level of abstraction of a program might affect its readability. Higher abstraction levels mean that more details are hidden, which could result in shorter lines. Shorter lines are stated to affect readability positively earlier in this section.

If the available abstraction levels in two programming languages differ, chances are that the overall readability also differs.

5.2.4 Verification

Methods for Software Verification

Software verification is an activity for checking that software works as desired [35]. In [36], software verification is divided into two areas, namely static and dynamic software verification.

Static software verification is the area of software verification methods which automatically verifies correctness of software without executing any code. In [37], a number of tools for static software verification are discussed. Many of them are mentioned to find problems like buffer overflows, memory leaks and redundant branch and loop conditions.

To find problems in the functionality, methods in the area of dynamic software verification are more likely to help. Dynamic software verification methods execute the code to verify that the program behaves as intended. Such methods are referred to as testing, or as stated in [38]: "Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior."

Testing

There are many aspects to test and verify correctness for in a program. An obvious aspect is the functionality, which should be tested to verify that the program returns the correct output for the provided input. Another aspect is the structure, which should be tested to verify that the program is implemented correctly. It is also interesting to test if the performance (execution time and memory consumption) of a program is satisfactory, which can be done using profiling software (as used in chapter 4 of this thesis).

The internal structure of a program can be tested using white-box testing [38], which denotes the set of testing techniques using information on how software is designed or coded.

Finite state machine based testing is a type of white-box testing described in [38]. It is performed by expressing the program to be tested as a finite state machine, and then verifying that the desired states and transitions are covered, for instance by adding code to produce output revealing which paths have been covered.

While white-box testing is good for finding design flaws, it does not find errors in the logic, which is the case for black-box testing.

Black-box testing [38] denotes techniques where a functionality of a program is verified by executing it with some input values and then checking that the results are correct. Unlike white-box testing techniques, coding and design details of the tested software are not considered.

Equivalence partitioning is a black-box testing technique described in [38] which is performed by dividing the set of possible input values into equivalence classes, which are subsets of the input set in which the elements have some desired properties in common. For instance if the possible input set is the set of all integers, a relevant equivalence class can be the set of negative numbers. The equivalence classes are chosen depending on what properties in the input one wants to test. The program is then tested with one or many elements from each equivalence class, to reveal if the program has problems with any equivalence classes.

Random testing is another black-box testing technique. It is performed by executing a program

with random input data and verifying the produced output data. This might seem too naive, for instance compared to equivalence partitioning described above, but because it might generally be simpler to use random testing, it might still be worth it.

Research has been done in [39] comparing the efficiency between equivalence partitioning (referred to as partition testing) and random testing. It was concluded that even though partition testing is slightly better at exposing faults, random testing still is the best choice when considering efficiency. To quote [39]: "By taking 20% more points in a random test, any advantage a partition test might have had is wiped out."

5.3 Evaluation

5.3.1 Maintainability

Type Safety

Since Feldspar does not allow any implicit type conversions due to the strong type system of Haskell, it is impossible to generate C code with such conversions. Since this is part of the language itself, not only the compiler, these restrictions always hold during the development process. The strong typing can be useful since it is very easy to see the types of a function's arguments and return value. Also, if a type error occurs, it will happen in the function that is causing it and not propagate through the functions and cause another error later on instead. However, a programmer might find it annoying that he has to be explicit about every conversion.

Consider the *DRS* test program (see section 3.3.2) which consists of many small functions. If one were to change the inner function `x_q` (see code in appendix A.2.1) to instead unsigned integers as arguments, this change has to be made at several different locations throughout the application chain.

```
x_q :: Data DefaultInt ->
     Data DefaultInt ->
     Data DefaultInt ->
     Data DefaultInt -> Data (Complex Float)
```

`x_q` is used in `r_bar`

```
r_bar :: Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt -> Data (Complex Float)
```

`r_bar` is used in `drs`

```
drs :: Data DefaultInt ->
     Data DefaultInt ->
```

```
    Data DefaultInt ->
    Data DefaultInt ->
    Data DefaultInt ->
    Data DefaultInt ->
    Data DefaultWord -> DVector (Complex Float)
```

This might not be of any risk at all, but the type system still forces you to be very explicit. In most cases, however, one can make the functions polymorphic (see section 2.1.2), and then make a specific wrapper function for the compilation. The polymorphic function will still be well-typed, since the polymorphic types will have constraints on them in order to preserve the strong typing.

In MATLAB, the situation is different. There is no need to be explicit about types, even though it is possible using the `assert` function [21]. Types are implicitly converted when needed, for example the expression `1 + 'm'` gives the result `110` (the meaning of `'m'` is its ASCII value). It is not obvious that this should be the case, one might instead expect that `1` should be converted to a character, and result in the concatenated string `'1m'`.

```
>> 1 + 'm'

ans =

 110
```

Example of an addition of `1` and the character `m` in MATLAB.

When generating code from a MATLAB function however, the C code becomes a strongly typed version of the function where no implicit conversions are made. This might be convenient because the functions can be very general during the development process, making them easy to change, and only in the end decide what types to use. It might also be dangerous, since it is possible to make mistakes in the compilation stage which may cause errors later. As an example, consider the test program *SqAvg* (see section 3.2.2), which takes a vector as input. If a vector with elements of different types are defined, MATLAB automatically converts the whole vector to the type with the highest class precedence (see [40]):

```
>> sqAvg([1,'m'])

ans =

      5941
```

Example of executing *SqAvg* with an input vector containing values of different types in MATLAB.

When compiling the function, the same thing happens. In this case it becomes a vector of type `char`, which may not be the expected result.

```
real_T sqAvg(const char_T in[2])
{
real_T d0;
int32_T i0;
```

```
d0 = 0.0;
for (i0 = 0; i0 < 2; i0++) {
    d0 += (real_T)in[i0] * (real_T)in[i0];
}
return d0 / 2.0;
}
```

The test program *SqAvg* compiled with: `codegen -config cfg -args {[1,'m']} sqAvq`

Debugging

MATLAB provides a capable source-level debugger [41] which lets you place breakpoints anywhere in the code, step through the code line-by-line and examine variable content. The debugger was used during the project, especially in the implementation of the *DRS* test program (section 3.3.2), and was very straight forward to use.

Feldspar currently has more limited possibilities for debugging. The simplest option is to use the trace function, which basically lets you output a string somewhere in a function.

There is also the GHCi debugger project, which aims to bring dynamic break points and inspection of intermediate values to GHCi [42]. However, no methods for debugging were actually used in the project, they are just mentioned to give some insight in the possibilities.

Documentation

Since Feldspar is embedded in Haskell, it is possible to use Haddock [43] for generating documentation from source files. Although it was not used during the project, the Feldspar API reference is generated using Haddock, and this reference was used extensively.

For MATLAB, one can use Doxygen [44]. It automatically extracts comments from the m-files in order to generate the documentation. No methods for generating documentation from MATLAB source files were used in the project.

5.3.2 Naive vs. Optimized

There are cases in both MATLAB and Feldspar where optimizations in the high-level code result in much better generated C code. In MATLAB, multiple function calls on vectors are not fused together as they are in Feldspar. See for instance the test program *SqAvg* (section 3.2.2), which first squares each element of the input vector and then computes the average of the squared vector. The test program generates two for-loops where the first squares each element and the second computes the sum:

```
real_T sqAvg(const real_T in[100000])
{
int32_T k;
static real_T x[100000];
real_T y;
```

```
for (k = 0; k < 100000; k++) {
    x[k] = in[k] * in[k];
}

y = x[0];
for (k = 0; k < 99999; k++) {
    y += x[k + 1];
}

return y / 100000.0;
}
```

The two for-loops in the generated code above can be reduced to one by exploiting the fact that the sum of squares of a vector can be expressed as a matrix multiplication of the vector and its transpose. Test program *SqAvg opt* generates a single for-loop where both the squares and sum are computed:

```
real_T sqAvg_opt(const real_T in[100000])
{
real_T y;
int32_T ix;
int32_T iy;
int32_T k;

y = 0.0;
ix = 0;
iy = 0;
for (k = 0; k < 100000; k++) {
    y += in[ix] * in[iy];
    ix++;
    iy++;
}

return y / 100000.0;
}
```

This is a clear case where it is important to know that using matrix operations in MATLAB is always better when it comes to performance. The programmer might very well find it more intuitive to implement it as in the first case, which generates very different code. *DotRev* is an example of the same problem, which also generates two for-loops, one for reversing the list and one for computing the scalar product. This is avoided in *DotRev opt* by manually computing the reverse and the scalar product at the same time inside a for-loop. However, this uses no benefits from MATLAB as a high-level language and is probably not the code the programmer wants to write.

In Feldspar, fusion is very useful when it comes to multiple function applications on vectors, since it guarantees that no intermediate vectors are created.

Another case is the DRS test program (see section 3.3.2) in the calculation of the pseudo-random sequence. Many programmers would probably find it intuitive to follow the specification, as in the initial DRS test program in MATLAB (see section 3.3.2). It uses a bit vector represented by a vector of integers in a for-loop.

```
intvector = [1, 1, 0, 1, 1, 0, 1, 0];
for i = 1...n
   intvector[i] = some calculation including other elements of intvector;
end
```

Pseudo code showing a for-loop with a vector containing integers as state.

As said in section 4.5.2, it is very bad practice to have a vector in the state of a for-loop like this in Feldspar, since it results in unnecessary copies of the vector in each loop iteration. This can be avoided by instead just having an integer in the for-loop state to represent the bit vector, which is the optimized version.

```
singleint = 2934857;
for i = 1...n
   singleint = some bit operations using singleint and i.
end
```

Pseudo code showing a for-loop with an integer as state.

This code looks very different compared to the naive version, and it is not trivial at all to understand that they even do the same thing (see the code for the test program DRS in appendix A.2). As seen in the results of the hard measures, the difference in performance is huge, the Feldspar version looping over a vector is so slow that its run on the C6670 simulator was canceled.

Both methods were implemented in MATLAB as well (see section 3.3.2), but the difference between these implementations was not nearly as big as for Feldspar, although significant. The non-optimized version still performs rather well compared to the optimized version.

5.3.3 Readability

Identifier Naming

When Feldspar code is compiled into C code, names for function arguments are not kept and all variables (including the ones corresponding to a Feldspar functions arguments) in the C code get default names depending on what they are used for.

MATLAB Coder on the other hand uses the variable names from the high-level code if possible. Sometimes new variables are needed and then default names are used like in Feldspar.

Since it obviously is more readable to get C code with variable names corresponding to the variables used in the high level code, MATLAB has a clear advantage over Feldspar in the matter of variable naming in the generated code. Also, comments in the MATLAB code are present in the generated C code as well, at the corresponding line. This is very useful since it enables the programmer to relate the MATLAB code to the C code. There is no such help for this in Feldspar, which can make it very hard to understand where the different parts of the generated C code comes from.

Line Length

For the high-level code, one might assume that it is obviously up to the programmer to decide how long the lines are. In both Feldspar and MATLAB, it is easy to break down large expressions into smaller and more readable ones. In MATLAB you have, for instance, the possibility of assigning sub-expressions to variables and using them in a larger expression, and in Feldspar you can either define new top level functions or use `where` and `let` clauses (see for instance the test program BinarySearch in section 3.2.2).

The interesting fact is that while the mentioned methods for Feldspar will result in the same generated code, this is not guaranteed for MATLAB. Below follows a simple example where a MATLAB expression first is assigned to a variable which is used later, and then where the expression is used directly.

#	MATLAB Code	Generated Code

1	`b = a * 10;`
	`out = 1;`
	`for i = 1:10`
	`  out = out*b;`
	`end`
		`b = 10.0 * a;`
		`out = 1.0;`
		`for (i = 0; i < 10; i++) {`
		`  out *= b;`
		`}`
		`return out;`

2	`out = 1;`
	`for i = 1:10`
	`  out = out*a*10;`
	`end`
		`out = 1.0;`
		`for (i = 0; i < 10; i++) {`
		`  out = out * 10.0 * a;`
		`}`
		`return out;`

Even though this particular example is not very interesting, since the first case (1) where the larger expression is broken down to sub-expressions generates better code, it is still an important fact that the structure of a program might affect the resulting generated code.

The fact above comes down to that a MATLAB programmer might need to consider performance when breaking down large expressions into sub-expressions, and since the number of identifiers per line is considered important for readability (see section 5.2.3), one might have to choose between performance and readability.

For the generated code, the most noticeable difference, considering line length, between Feldspar and MATLAB is array indexing. Because of the C code representation of Feldspar vectors (see section 2.1.5), array indexing tends to yield rather long lines compared to MATLAB, where the usual `array[index]` notation is used in the generated code. For expressions with multidimensional vectors, this means that Feldspar will suffer from very long lines, resulting in less readability (according to section 5.2.3).

Uneccessary Code

In Feldspar, the problem with vector updating (like swapping two elements) using the vector library also results in problems with readability in the generated code. Consider a function which swaps

to elements in a vector using the vector library.

```
swap :: DVector DefaultInt -> Data Index -> Data Index -> DVector DefaultInt
swap v i1 i2 = permute (\l i -> (i == i1) ? (i2, (i == i2) ? (i1, i))) v
```

This generates the following code:

```
for(i4 = 0; i4 < length(in0); i4 += 1)
{
    uint32_t w5;

    if((i4 == in1))
    {
     w5 = in2;
    }
    else
    {
     if((i4 == in2))
     {
         w5 = in1;
     }
     else
     {
         w5 = i4;
     }
    }
    at(int32_t,(* out3),i4) = at(int32_t,in0,w5);
}
```

To swap the elements, the whole vector has to be looped through, looking for the indices to be swapped. This could of course have been written much simpler. See the code below.

```
copyArray(out3, in0);
at(int32_t,(* out3),in2) = at(int32_t,in0,in1);
at(int32_t,(* out3),in1) = at(int32_t,in0,in2);
```

Feldspar can generate the code above if the *core language* is used (see section 2.1.2 for a brief description and the function `mmse_eq1_core` in appendix A.4.1). This however results in a lower level of abstraction as well as unnecessary array copying.

In MATLAB, strange and unecessary things were sometimes generated in the C code. In the test program *SqAvg opt* (see section 5.3.2), two completely unnecessary counters were defined and incremented in each iteration of the for-loop.

Levels of Abstraction

Most problems can be expressed at many levels of abstraction in Feldspar and MATLAB, which makes the area hard to evaluate. Though one interesting difference between the languages has been

noticed when coding at high levels of abstraction. Below follows an example of a mathematical specification to the test program *SqAvg* (see section 3.2.2), together with implementations in both languages. The specification is needed as a reference to motivate the levels of abstraction of the implementations.

Mathematical specification of the *SqAvg* test program:

$$sqAvg(v) = \frac{\sum_{i=0}^{|v|} v_i^2}{|v|}$$

As seen in the specification above, all elements of the vector `v` are squared and summed before the result is finally divided by the length of `v`.

Feldspar implementation:

```
sqAvg :: DVector Float -> Data Float
sqAvg v = fold f 0 v / (length v)
    where f s n = s + n**2
```

This is an alternative implementation of the *SqAvg* test program to show how the higher-order function `fold` can be used. `fold` takes a function, a start value and a vector as arguments. It applies the function to the start value and the first element of the vector, then it feeds the function with the result and the second element of the vector, and so on throughout the vector. The final result of the function is returned.

MATLAB implementation:

```
function out = sqAvg_opt(in)

out = (in*in.')/length(in);

end
```

In the MATLAB implementation above, the input vector `in` is seen as a matrix. `in` is multiplied with its transpose using matrix multiplication to yield the squaring and summation.

Both the Feldspar and the MATLAB implementation are on higher abstraction levels than the mathematical specification, in the sense that details are hidden away from the programmer. The counter variable `i` in the specification is implicit in Feldspar because of `fold` and in MATLAB because of implicit indexing in the matrix multiplication.

Programming on high abstraction levels like above can make an implementation differ much from a mathematical specification, which is considered to affect the readability negatively in this thesis. On the other hand, hiding details by programming at higher levels of abstraction usually means shorter lines which is considered good for the readability (as stated in section 5.2.3). High abstraction levels can thus affect readability both negatively and positively.

This example shows that Feldspar probably has more possibilities of coding at high abstraction levels than MATLAB, because Feldspar's higher order functions can probably be used to solve a larger set of problems than matrix arithmetic.

5.3.4 Verification

Plotting

A program's functionality can be tested and verified using plotting functionalities. Plotting involves printing a discrete series of values on the screen, preferably with a line connecting the values, and preferably with possibilities of zooming, tracing, calculating derivatives of the curves etc.

Visualizing the output of a program by plotting it can be of great help to assert that the program behaves as it should. One can easily plot ranges of output values from relevant ranges of input values and literally get a picture of the correctness of the tested function. Plotting is supposedly more useful for black-box than white-box testing techniques, because it can express input-output relations of a program. For white-box testing, methods like manual analysis or adding code for structure checking are used, and plotting the results is probably not relevant.

Plotting can be done in MATLAB using the `plot` function, which makes it is possible to plot elements of vectors and matrices and provide values for scaling the axes. The `plot` function in MATLAB can only handle 2D plotting, but there are also functions for plotting 3D lines and surfaces, for example `plot3` and `surf`.

Example of using `plot` in MATLAB:

```
>> y = sin((1:100000)*2*pi*440/100000);
>> plot((1:500),y(1:500))
```

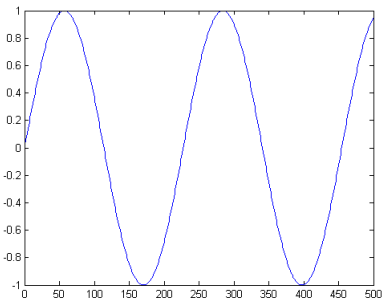

Figure 5.1: A sinus signal plotted in MATLAB.

It is also possible to plot data from Feldspar programs by evaluating the Feldspar program into Haskell values (using the `eval` function), and then use any desired Haskell graphics library to print the values on the screen. In this thesis, Gnuplot was used as an example [45].

The following example shows how to use Gnuplot with Feldspar:

```
import Feldspar
import Feldspar.Vector
import Feldspar.Compiler
import Graphics.Gnuplot.Simple

a :: DVector Float
a = indexed 100000 (\i -> sin(440*(2*pi)*(i2f i)/100000))
```

Then call `plotList` in the interpreter:

```
*Main> plotList [] (eval (take 500 v))
```

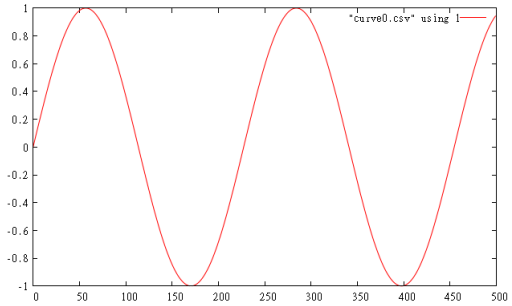

Figure 5.2: A sinus signal from Feldspar plotted with Gnuplot.

Since plotting is included in the MATLAB environment, it was easier to use than in Feldspar.

Random Testing using QuickCheck

QuickCheck is a lightweight random testing tool for Haskell [46]. It requires the user to formally express the properties to be tested, and gives the user great freedom in deciding how to generate random input data for the tested program.

Since Feldspar is embedded in Haskell, properties can be expressed using any Feldspar functions, as long as `eval` is called to bring Feldspar values out to Haskell. In a similar way, random generators constructing Feldspar values for input to the tested function can be defined by bringing Haskell values into Feldspar (using `vector` in the example below).

A simple example where QuickCheck has been used to test the *BubbleSort* test program (section 3.2.2) is presented below. Details of this example are out of the scope of this thesis, see the QuickCheck manual for more information [47].

```
import qualified Feldspar as FS
import qualified Feldspar.Vector as FSV
import qualified Test.QuickCheck as QC
import Control.Monad
import BubbleSort

instance Show (FSV.DVector FS.Int32) where
show v = show $ FS.eval v

instance QC.Arbitrary (FSV.DVector FS.Int32) where
arbitrary = liftM FSV.vector (QC.listOf QC.arbitrarySizedBoundedIntegral)


prop_BSort xs = isSorted bubbled && length xs' == length bubbled
where isSorted :: [FS.Int32] -> Bool
      isSorted [] = True
      isSorted (x:rest) = x <= head rest && isSorted rest
```

```
    bubbled = FS.eval $ bubbleSort xs
    xs' = FS.eval xs
    types = xs :: FSV.DVector FS.Int32


*Main> QC.quickCheck prop_BSort
+++ OK, passed 100 tests.
```

In the example above, `prop_BSort` defines a property which must hold for `bubbleSort` and sorting algorithms in general. The property says that the output list of `bubbleSort` should be sorted and of the same length as the input list. In this example, the `bubbleSort` function from the test programs is located in the module `BubbleSort`.

MATLAB has no such general tool for random testing; thus one has to implement test functions manually if visual verification is not enough (as explained in the plotting section).

Plotting and random testing are useful for different tasks and one of the methods cannot replace the other. In some cases, when the exact desired behaviour of a function is not known, it can be difficult to write QuickCheck properties to test. Then plotting can be used to analyze the results to see if they are reasonable.

5.4 Survey

5.4.1 Method

A brief survey was composed and handed out as part of the productivity evaluation, with the purpose of getting personal opinions about Feldspar and MATLAB from people with varying experience. The survey consisted of 5 problems with code examples from one or both languages. Each problem is followed by a set of questions. Some questions were answered by selecting the number corresponding to the opinion about the question, and some were answered by writing the opinion in free text. One question was also answered by writing Feldspar and MATLAB code. The survey can be found in appendix C together with the results.

The problems and questions were chosen to evaluate readability (question 1), preferred abstraction level (question 2) and maintainability (question 3). Language references to Feldspar and MATLAB were added at the end of the survey to let the participants look up the functions used in the questions.

5.4.2 Results

There were only 10 participants, but since the survey was not a significant part of the productivity evaluation, this was not a problem. Because of the few participants, one should be careful not to draw any general conclusions from the results.

One observation is that the opinions are affected by their experiences with MATLAB and functional programming. Almost everyone who answered most questions in favour to one language also had higher experience in that language. This is mainly seen in the sub-questions to questions 1, 2 and 3, where the participants were asked how hard it was to understand a function, how intuitive a

function was or how easy it was to change a function. The answers were observed to almost always reflect the experience of the participants.

In question 2 b, the participants were asked if they preferred explicit or implicit counter variables in loops. All participants with more experience in functional programming than MATLAB preferred implicit counters, and the participants with more MATLAB experience had evenly distributed varying opinions. This might indicate that some MATLAB users would actually prefer using Feldspar because of its possibilities for programming on high abstraction levels (discussed in section 5.3.3).

Question 3 was meant to examine the opinions about maintainability, and more specific the differences in type safety in Feldspar and MATLAB. 5 of the 10 participants pointed out that they preferred Feldspars strong type system, because of reasons like having a language with strong type signatures makes software design more secure. 3 out of these 5 had more experience in MATLAB than in Feldspar, and 1 had equally high experience in both languages. This indicates that a strong type system like the one of Feldspar might actually be preferred, even by experienced MATLAB users. Though it should be noted that it is possible but optional to enforce what types a function uses in MATLAB, as discussed in section 2.2.2.

Chapter 6

Discussion

6.1 Fundamental Differences

There are many fundamental differences between Feldspar and MATLAB. Perhaps the most significant difference is the purpose of the languages. MATLAB focuses on efficient vector and matrix calculations in many different academic and industrial areas. It includes many built-in mathematical functions and useful visualization tools which makes it a good language and environment for modelling and verifying algorithms. Code generation is thus not the main purpose of MATLAB, but rather an extension. Feldspar on the other hand has a more narrow aim towards being efficient for DSP and real-time algorithm design for embedded systems. The focus around embedded systems has made C code generation one of the main aspects of the language. By just considering the languages purposes, the broader purpose of MATLAB can be an advantage if generality is desired, and Feldspar could possibly prove to have an advantage for DSP algorithm design in the future when the language is more mature.

Another important difference is that MATLAB features a complete IDE for coding, debugging, executing, visualizing programs etc. Feldspar lacks this, but can on the other hand use many existing Haskell tools, for instance for visualization or debugging (see chapter 5), with slightly more effort required to set up.

Finally it should be noted that Feldspar is open source, which gives its users insight in the implementation details and even the possibility of modifying the language. Being open source, Feldspar is free, compared to MATLAB which costs money to use.

6.2 Observations from the Hard Measures

It is very important not to forget that this thesis has only compared Feldspar and MATLAB. There is no reference for how the desired results of the performance measures should be. Even if the implementation in one language shows much better performance than the implementation in the other language, it does not mean that the best result is good compared to an ideal case.

The generated C code from MATLAB and Feldspar was run on two different platforms (see section 4.1) in order for the results to be more platform independent. This showed to be a good choice, since some results differed between the platforms (see for instance section 4.6.4).

MATLAB showed much lower execution times in many cases, including all DSP test programs, which perhaps are of most interest since they are more realistic. However, the reason for Feldspar's poor performance seems to reduce into three specific problems. First, there is the problem of having a vector as state in the `forLoop` function (described in section 4.5.2), which introduces a lot of unnecessary copying which both takes time and space. Secondly, the results showed that explicitly setting the lengths of input vectors was very important for performance (see results for *MMSE EQ1*, *TwoFir* and *ChEst* in section 4.6). Finally, in the *DRS* test program (section 3.3.2), the creation of three identical prime tables in the generated C code (see section 4.6.7) resulted in slower execution times and higher memory consumption.

Feldspar performed better than MATLAB in some of the small test programs because of fusion (see section 2.1.4). This contributes to lower memory consumption (see section 4.7) because no intermediate vectors have to be stored. It can also contribute to reduced execution time, since several for-loops performing different computations can be merged to one.

Fusion in Feldspar relies heavily on the guarantee that no side effects occur. To implement fusion in MATLAB would probably be much more complicated, since side effects are allowed. If two function applications on a vector were to be fused in MATLAB, and the functions have side effects, the ordering of the effects in a fused version might not be the same as in a non-fused version. This might change the result, which probably is the reason why fusion does not exist in MATLAB. However, in test programs such as *SqAvg*, it was observed that reformulating problems in terms of vector and matrix operations can be used to obtain a fusion-like behaviour (see section 3.2.2).

An interesting observation was made when using wrappers to explicitly set the lengths of input vectors. This resulted in greatly increased performance for some examples, especially *MMSE EQ1* (see section 4.6.9). In this case it is probably mainly because of the large amount of C code that disappeared in the wrapped version (see 4.8). In the test program *DotRev*, however, it was observed that the optimized MATLAB implementation performed better than the Feldspar version. The C code generated from Feldspar was manually edited to have fixed-size input vectors, which resulted in execution time equal to the optimized MATLAB version. This means that the fixed-size input vectors is of major importance when it comes to performance. It might have been more fair to use the second memory allocation method described in section 2.2.3 for MATLAB, but this would on the other hand have generated checks for index-bounds etc, which Feldspar would not. It is not obvious which method would have been best for the comparison, but at least one should know how big the differences are.

For actually using the generated C code in this project, MATLAB proved to induce much less effort than Feldspar. MATLAB did not generate any code that was not compatible with the compilers used, as Feldspar did with complex arithmetic (see section 4.5.1). Also, since the necessary memory was allocated on the stack inside the functions (with dynamic memory allocation turned off), there was no need for allocating this manually, as was the case in Feldspar (see sections 2.1.5 and section 4.5.3).

It was observed that MATLAB Coder was able to simplify expressions in some cases which resulted in better C code (see *AddSub* and *TransTrans* in section 3.2. In these cases, no code was generated at all, which of course is the desired result. Feldspar did not manage to completely remove these expressions, as seen in section 4.6.6.

6.3 Observations from the Soft Measures

For the naive vs optimized part, there were two major situations in MATLAB and three in Feldspar where interesting observations were made. In MATLAB, it was sometimes possible to achieve a

fusion-like behaviour, i e. no unnecessary introduction of an intermediate vector. It was possible by either reformulating the problem using vector or matrix operations (see *SqAvg* in section 3.2.2), or by doing it manually by writing C-like code (see *DotRev opt* in section 3.2.2). While the latter of course works, it takes the programming to a lower level of abstraction, which should not be preferred. Reformulating the problems to use vector and matrix operations might not be trivial, if possible at all, so it would be very convenient if the same code was generated when using both methods.

In the naive vs optimized section (5.3.2), interesting things concerning Feldspar were observed in cases where vectors were used in the state of `forLoop` functions and vector elements were to be updated. When using the vector library, the only way to update the elements was to change the indexing function of the vector, which generates code with a lot of conditionals (as shown in section 5.3.3). Slightly better C code could be generated by instead using core arrays, but this also forces the programmer to program at a lower level of abstraction than that of the vector library.

Also, in the *DRS* test program (see section 3.3.2) it was possible to avoid having a vector in the state of a `forLoop` function, by instead using an integer in the state together with bit operations. The version with a vector in the state performed so badly that the test runs were canceled for that particular case.

Both MATLAB Coder and the Feldspar compiler generated well structured C code, as discussed in section 5.3.3. However, MATLAB have an advantage in readability by the fact that variable names are the same in the generated code as those in the MATLAB code. Also, comments in the MATLAB code are present in the generated code, which makes it easier to relate the generated code to the high-level code. In Feldspar however, especially for larger programs, it can be almost impossible to figure out where the code actually came from. There were also cases where MATLAB introduced unnecessary variables in the generated code (see the optimized version of the test program *SqAvg* in section 5.3.3).

Feldspar's possibilities of arbitrarily restructuring a program using `where` and `let` clauses (see section 5.3.3) in order to improve readability, but without changing the generated code was considered good. In MATLAB, the structure proved to be important for the resulting C code. This might be the intuitive way for programmers used to imperative languages, but regarding readability it might not be the best way.

As discussed in section 5.3.3, the higher-order functions of Feldspar give the programmer possibilities of writing a program at higher abstraction level than a corresponding mathematical specification. This can be both good and bad for readability, and usually results in less Feldspar code. In MATLAB, a higher abstraction level could be reached if the problem was reformulated to use matrix operations. Even though this can be done in many cases, the method is much less general than the higher-order functions of Feldspar.

Regarding type systems (discussed in section 5.3.1), it is not easy to say which method is best. In both languages it is possible to design functions with generalized types, and then either define the types in a separate wrapper function for compiling (Feldspar), or when invoking the compiler (MATLAB). This can be seen as a quite similar method, and was considered to require equal amount of effort. However, functions can be made even more general in MATLAB, where functions can be applied to both vectors/matrices and scalars. An observation was that MATLAB allows defining vectors with elements of different types where MATLAB automatically converts the vector to the type with highest class precedence (as discussed in section 5.3.1). This might be confusing for the programmer if he is not aware of how MATLAB handles these conversions. In Feldspar, no implicit type conversions can occur, which provides more safety but might be less user friendly.

Chapter 7

Conclusions

7.1 The Status of Feldspar

One objective of this thesis was to evaluate the current status of Feldspar and inform Ericsson about it. This section contains a summary of observed properties.

Programming in Feldspar is much like programming in Haskell, and many features of Haskell may be used. There are currently some problems which have to be solved (see section 6.2) in order to reach an overall performance comparable to MATLAB. However, results have shown that in test programs where these problems do not apply, fusion contributes to performance equal to or higher than MATLAB. According to the Feldspar developers, there are ideas about how to solve these problems, which makes Feldspar a promising language in future DSP software development.

To compile and run the generated C code is sometimes not trivial. The compiler currently has no option for generating ANSI C code, which means that many compilers will not be able to compile the code. However, it should be very easy for the developers to add support for ANSI C, which does not make this a very big problem. Another problem is the memory handling. The programmer manually has to figure out how much memory the program needs and allocate it. It should be noted that Feldspar is not intended to be like this, and that functionality for giving information about the memory allocation has recently been added (although not documented).

Many libraries that would be useful for DSP programming, such as matrix operations and fixed-point arithmetic have currently rather limited functionality. The developers are currently working on improving the libraries, and it has been observed that the support for fixed-point arithmetic has improved during this thesis project.

7.2 Feedback to Developers

One objective of this thesis was to provide feedback to the developers of Feldspar and MATLAB. The following sections contain feedback based on observations from the performance and productivity evaluations in chapters 4 and 5. Also, the experiences from the authors of this thesis has been taken into account.

7.2.1 Feldspar

- The performance of Feldspar would greatly increase in many situations if the `forLoop` function with vectors in the state generated code without unnecessary array copying. As discussed in section 4.5.2, this proved to be a big problem. It would also be great if a function similar to a while-loop was included, since the `forLoop` function needs to run all of its iterations.
- Explicitly setting the lengths of input vectors (as discussed in section 2.1.5) proved to be of huge importance in some cases (see the results of *TwoFir*, *ChEst* and *MMSE EQ* in section 4.6). It both helps the Feldspar compiler to generate less code, as well as the C compiler to generate faster code. This should be more documented and its importance could be more clearly stated. Also, it might be good to make `wrap` able to wrap multi-dimensional vectors of arbitrary lengths.
- It was very time consuming to manually analyze the generated C code to figure out the structure of the array `mem` (the problem is explained in detail in section 4.5.3). Consider automatic generation of this information.
- Even though Feldspar aims to support many different platforms in the future, it might be good to have a predefined option for generating ANSI C code. ISO C99 was not fully supported by any of the compilers used in this thesis, which made generated C code from Feldspar hard to use (as stated in section 4.5.1).
- In the generated C code for the DRS test program (see section 3.3.2), a large table of prime numbers was defined more times than necessary. This increased the execution time significantly.
- It would be very useful to somehow generate information about the relation between Feldspar code and its generated C code. The generated code from MATLAB uses the same variable names as the high-level code (discussed in section 5.3.3). It was impossible to figure out the origin of the C code generated from larger Feldspar programs. Variable names revealing variables' origins would be preferred.
- The current method of indexing in arrays in the generated C code leads to long lines. This is bad for the readability of the generated code as discussed in section 5.3.3. The method of indexing could perhaps be improved to raise the readability.
- It would be nice if expressions could be further simplified by the compiler. As seen in the test programs *AddSub*, *RevRev* and *TransTrans* (see section 4.6.6), C code doing unnecessary things was generated.

7.2.2 MATLAB

- There were many occasions where MATLAB suffered from introducing intermediate arrays in the C code, where all calculations instead could have been done in one array (see the example in section 5.3.2). The possibilities of introducing something similiar to fusion should be investigated to improve performance.
- Problems can often be reformulated to use matrix arithmetic, which can result in better generated C code (see the test program *SqAvg* in section 3.2.2). For such cases it would be preferred to get the same good generated C code.
- For many languages, not only Haskell, there is an implementation of QuickCheck. That might be a useful tool also for MATLAB. The possibilities for this should be investigated. Random testing and QuickCheck are discussed in sections 5.2.4 and 5.3.4.

- It was noted that MATLAB allows vectors containing different types to be defined. Such vectors are converted to the type with the highest class precedence. This may be confusing for the programmer, and a warning would be preferred if such a vector is used with MATLAB Coder (see section 5.3.1).
- In some test programs, MATLAB generated unnecessary code. See for instance the optimized version of the test program *SqAvg*, described in section 5.3.3 together with its generated C code. This could perhaps be investigated and improved.

7.3 Future Work

This thesis did not investigate how well the languages performed against handwritten C code, because the objectives only stated that a relative comparison between the languages should be made. It would be interesting to evaluate how the languages perform relative to an ideal solution.

Fixed-point arithmetic was not evaluated in this thesis because of the limited support for it in Feldspar. When Feldspar supports fixed-point arithmetic at a level closer to MATLAB, the area would be very interesting to evaluate. The same goes for generation of multi-core code which was currently not supported by any of the languages.

In this thesis, the C code was generated without any hardware adaptions such as intrinsics. This in order to not be platform specific, because Feldspar currently only supports one embedded platform with no floating-point support. When Feldspar supports more interesting platforms, performance can be measured on other platforms than in this thesis to compare the languages in terms of portability.

The survey only made up a small part of the soft measures evaluation, because it would have been hard to do it in a scientific manner when so few people knew about Feldspar. A more scientific survey would be an interesting future project.

When the grave problems observed in this thesis are solved, new problems at a finer level might arise. A new evaluation would be preferred at that time.

Appendix A

Appendix: Code

A.1 Small Test Programs

A.1.1 Feldspar

```
import qualified Prelude
import Feldspar
import Feldspar.Compiler
import Feldspar.Vector
import Feldspar.Matrix
import PlatformAnsiC

-- Two fir filters in series
twoFir :: DVector Float -> DVector Float -> DVector Float -> DVector Float
twoFir b1 b2 = convolution b1 . convolution b2

-- twoFir without fusion to avoid repeated computations.
twoFir2 :: DVector Float -> DVector Float -> DVector Float -> DVector Float
twoFir2 b1 b2 = convolution b1 . force . convolution b2


-- Compile version with static input vector sizes.
twoFir2_compile
    :: Data [Float]
    -> Data [Float]
    -> Data [Float]
    -> DVector Float
twoFir2_compile b1 b2 v =
    twoFir2 (unfreezeVector' 20 b1) (unfreezeVector' 20 b2) (unfreezeVector' 12880 v)

-- Squared average of vector
sqAvg :: DVector Float -> Data Float
sqAvg v = sum (v .* v)/i2f (length v)

-- Scalar product of vector and reverse of vector
dotRev :: DVector Int32 -> Data Int32
```

```
dotRev v = scalarProd v $ reverse v

-- Reverse of reverse
revrev :: DVector Int32 -> DVector Int32
revrev = reverse . reverse

-- Transpose of transpose
transtrans :: Matrix Int32 -> Matrix Int32
transtrans = transpose . transpose

-- Adding and subtracting by one
addsub :: DVector Int32 -> DVector Int32
addsub = map (\x -> x-1) . map (+1)


-- Binary search
binarySearch :: Data DefaultWord -> DVector DefaultWord -> Data Index
binarySearch key v = fst $ forLoop iters (0,len-1) f
  where len = length v
        iters = ceiling $ logBase 2 $ i2f len
        f _ (low,high) =
            (v!d == key) ? ((d, d), ((key < v!d) ? ((low, d-1), (d+1, high))))
                where d = (low + high) `div` 2

-- Identity matrix
idMatrix :: Data DefaultWord -> Matrix DefaultWord
idMatrix n = indexedMat n n (\i j -> b2i (i == j))

-- Slice of matrix
sliceMatrix :: (Data Index, Data Index) ->
               (Data Index, Data Index) ->
               Matrix DefaultWord -> Matrix DefaultWord
sliceMatrix x y = colgate y . map (colgate x)
  where
    colgate (x1, x2) = drop (x1-1) . take x2


-- Slice of identity matrix
idmSlice :: (Data Index, Data Index) ->
            (Data Index, Data Index) ->
            Data DefaultWord -> Matrix DefaultWord
idmSlice x y n = sliceMatrix x y $ idMatrix n

-- Duplicate list
duplicate :: DVector Int32 -> DVector Int32
duplicate v = v ++ v

-- ... with a forLoop and 2 writes per iteration
duplicate2:: DVector Int32 -> DVector Int32
duplicate2 v = unfreezeVector $
                  forLoop l
                          start
                          (\i s -> setIx (setIx s (i + l) (getIx a i)) i (getIx a i))
  where l = length v
        a = freezeVector v
        start = freezeVector $ replicate (l*2) 0
```

```
-- Bubble sort (setIx)
bubbleSort :: DVector Int32 -> DVector Int32
bubbleSort v = forLoop len v inner

  where len = length v
        inner i nv = forLoop (len-1) nv bubble
        bubble j nv' =
            (nv'!j > nv'!(j+1)) ?
            ((unfreezeVector $ swap1 (freezeVector nv') j (j+1)), nv')
        swap1 a i1 i2 = setIx (setIx a i1 (getIx a i2)) i2 (getIx a i1)


-- Bubble sort (permute)
bubbleSort2 :: DVector Int32 -> DVector Int32
bubbleSort2 v = forLoop len v inner

  where len = length v
        inner i nv = forLoop (len-1) nv bubble
        bubble j nv' = (nv'!j > nv'!(j+1)) ? (swap nv' j (j+1), nv')
        swap v i1 i2 = permute (\l i -> (i == i1) ? (i2, (i == i2) ? (i1, i))) v


-- Convolution (from Examples/Math/Convolution.hs)
convolution :: DVector Float -> DVector Float -> DVector Float
convolution kernel input = map ((scalarProd kernel) . reverse) $ inits input



-- Bit reversal (from Example/Math/Fft.hs)
bitRev :: Type a => Data Index -> Vector (Data a) -> Vector (Data a)
bitRev n = pipe riffle (1...n)

pipe :: (Syntactic a) => (Data Index -> a -> a) -> Vector (Data Index) -> a -> a
pipe = flip.fold.flip

rotBit :: Data Index -> Data Index -> Data Index
rotBit 0 _ = error "k should be at least 1"
rotBit k i = lefts .|. rights
  where
    ir = i >> 1
    rights = ir .&. (oneBits k)
    lefts  = (((ir >> k) << 1) .|. (i .&. 1)) << k

riffle k (Indexed l ixf Empty) = indexed l (ixf.rotBit k)

oneBits n = complement (allOnes << n)
allOnes = complement 0
```

A.1.2 MATLAB

```
%%%% SMALL TEST PROGRAMS
```

```
% AddSub
% codegen -c -config cfg -args {zeros(1,100000,'int32')} addsub

function in = addsub(in) %#codegen

in = (in + 1) - 1;

end


% BinarySearch
% codegen -c -config cfg -args {zeros(1,100000,'int32'), int32(0)} binarysearch

function o = binarysearch(x,key) %#codegen

a = int32(1);
b = int32(length(x));

o = int32(b+1);

while a <= b
    mid = ceil((a+b)/2);
    if(key < x(mid))
       b = mid-1;
    elseif(key > x(mid))
       a = mid+1;
    else
       o = mid;
       break;
    end
end

end


% BinarySearch_opt
% codegen -c -config cfg -args {zeros(1,100000,'int32'), int32(0)} binarysearch_opt

function o = binarysearch_opt(x,key) %#codegen

a = int32(1);
b = int32(length(x));

o = int32(b+1);

while a <= b
    mid = idivide(a+b,int32(2),'ceil');
    if(key < x(mid))
       b = mid-1;
    elseif(key > x(mid))
       a = mid+1;
    else
       o = mid;
       break;
    end
end
```

```
end


% BitRev
% codegen -c -config cfg -args {(1:16)} bitrev

function in = bitrev(in) %#codegen

    in = bitrevorder(in);

end


% BubbleSort
% codegen -c -config cfg -args {zeros(1,1000,'int32')} bubblesort

function x = bubblesort(x) %#codegen

len = length(x);

for i = 1:len
    swapped = false;
    for j = 1:len-1
       if x(j) > x(j+1)
           swapped = true;
           temp = x(j);
           x(j) = x(j+1);
           x(j+1) = temp;
       end
    end
    if ~swapped
       break;
    end
end


% BubbleSort_opt
% codegen -c -config cfg -args {zeros(1,1000,'int32')} bubblesort_opt

function x = bubblesort_opt(x) %#codegen

len = length(x);

swapped = true;
i = 0;
while swapped && i < len
    swapped = false;
    for j = 1:len-1
       if x(j) > x(j+1)
           swapped = true;
           temp = x(j);
           x(j) = x(j+1);
           x(j+1) = temp;
       end
    end
```

```
    i = i + 1;
end


% DotRev
% codegen -c -config cfg -args {zeros(1,100000,'int32')} dotrev

function out = dotrev(in) %#codegen

out = dot(fliplr(in),in);

end


% DotRev_opt
% codegen -c -config cfg -args {zeros(1,100000,'int32')} dotrev_opt

function out = dotrev_opt(in) %#codegen

out = zeros(class(in));
n = numel(in);
nd2 = floor(numel(in)/2);
for k = 1:nd2
    out = out + in(k)*in(n-k+1);
end
if 2*nd2 ~= n
    out = out + in(nd2+1)*in(nd2+1);
end

end


% Duplicate
% codegen -c -config cfg -args {zeros(1,100000,'int32')} duplicate

function o = duplicate(x) %#codegen

o = [x x];

end


% IdMatrix
% codegen -c -config cfg idmatrix

function o = idmatrix() %#codegen

o = eye(1000,'int32');

end


% IdMatrix_opt
% codegen -c -config cfg idmatrix_opt

function o = idmatrix_opt() %#codegen
```

```
o = zeros(1000,'int32');

for k=1:1000
    o(k,k) = 1;
end

end


% RevRev
% codegen -c -config cfg -args {zeros(1,100000,'int32')} revrev

function in = revrev(in) %#codegen

    in = fliplr(fliplr(in));

end


% SliceMat
% codegen -c -config cfg -args {int32(0), int32(0), int32(0), int32(0)} slicemat

function o = slicemat(x1,x2,y1,y2) %#codegen

coder.varsize('o', [1000 1000], [1 1]);

m = eye(1000,'int32');
o = m(x1:x2,y1:y2);

end


% SqAvg
% codegen -c -config cfg -args {zeros(1,100000,'double')} sqavg

function out = sqavg(in) %#codegen

out = sum(in .* in)/length(in);

end


% SqAvg_opt
% codegen -c -config cfg -args {zeros(1,100000,'double')} sqavg_opt

function out = sqavg_opt(in) %#codegen

out = (in*in.')/length(in);

end


% TransTrans
function m = transtrans(m) %#codegen
```

```
m=m'';

end


% TwoFir
% codegen -c -config cfg
    -args {zeros(1,100000,'double'),
           zeros(1,10,'double'),
           zeros(1,10,'double')} twofir

function in = twofir(in,k1,k2) %#codegen

in = filter(k2,1,filter(k1,1,in));

end
```

A.2 DRS

A.2.1 Feldspar

```
import qualified Prelude as P
import Feldspar
import Feldspar.Vector
import Feldspar.Compiler
import PlatformAnsiC
import Control.Arrow ((***)) -- To generate c_n, the preudo random sequence numbers.


-- Computes the demodulation reference signal sequence.
-- n_cs     - Number of subcarriers
-- v        - Base sequence number
-- cs       - Cyclic shift
-- csf      - Cyclic shift field number
-- n_cid    - Cell ID number
-- delta_ss - Parameter configured by higher layers
drs :: Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultWord -> DVector (Complex Float)
drs n_cs v cs csf n_cid delta_ss slot = zipWith (*)
                        (indexed (i2n n_cs)
                                 (\i -> cis ((i2n i) * a)))
                        (indexed (i2n n_cs)
                                 (\i -> r_bar (i2n i) n_cs v n_cid delta_ss))
  where a = alpha slot cs csf n_cid

drs_all_slots :: Data DefaultInt ->
```

```
                Data DefaultInt ->
                Data DefaultInt ->
                Data DefaultInt ->
                Data DefaultInt ->
                Data DefaultInt -> Vector (DVector (Complex Float))
drs_all_slots n_cs v cs csf n_cid delta_ss =
    indexed 20 $ \slot -> drs n_cs v cs csf n_cid delta_ss slot

--Element n of reference signal sequence (r_uv_alpha) - 3GPP 5.5.1
r_n_r_bar :: Data Float ->
             Data Float ->
             Data (Complex Float) -> Data (Complex Float)
r_n_r_bar n alpha_in r_bar_in = r_bar_in * cis (alpha_in*n)


--Base sequences of length 3Nsc or larger - 3GPP 5.5.1.1

r_bar :: Data DefaultInt ->
         Data DefaultInt ->
         Data DefaultInt ->
         Data DefaultInt ->
         Data DefaultInt -> Data (Complex Float)
r_bar n m_sc v n_cellid delta_ss = x_q (n `rem` nzc) nzc v (u n_cellid delta_ss)
    where nzc = n_rs_zc m_sc


-- Function: Calculates the parameter f_ss_pucch from the Cell ID according to
-- 3GPP 36.211 section 5.5.1.3
-- Application example: eval (f_ss_pucch 2232131::Data Int)
f_ss_pucch :: Data DefaultInt -> Data DefaultInt
f_ss_pucch _N_cell_ID = _N_cell_ID `rem` 30

-- Function: Calculates the parameter f_ss_pusch from the Cell ID according to
-- 3GPP 36.211 section 5.5.1.4
-- Application example: eval (f_ss_pucch 2232131::Data Int)
f_ss_pusch :: Data DefaultInt -> Data DefaultInt-> Data DefaultInt
f_ss_pusch _N_cell_ID delta_ss = (f_ss_pucch _N_cell_ID + delta_ss)`rem` 30

-- Function: Calculates the parameter f_gh according to 3GPP 36.211 section 5.5.1.3
-- Application example: eval f_gh
f_gh :: Data DefaultInt
f_gh  = 0 -- Group hoppnig is dissabled in this implementation.

-- Function: Calculates the Group hoppnig parameter u from the Cell ID according
-- to 3GPP 36.211 section 5.5.1.3
-- Application example: eval (u 12 0 )
u :: Data DefaultInt -> Data DefaultInt -> Data DefaultInt
u _N_cell_ID delta_ss = (f_gh + f_ss_pusch _N_cell_ID delta_ss)`rem` 30

x_q2 :: Data DefaultInt ->
        Data DefaultInt ->
        Data DefaultInt ->
        Data DefaultInt -> Data (Complex Float)
x_q2 m nzc v u = cis a
   where
      q_mod=rem (q*m*(m+1)) (2 *nzc)
```

```
      nsc_scaling=div 32767 nzc
      a = i2f (65536 - q_mod * nsc_scaling)
      q = ((q_barx 1) + 1)+v*(1-(2*(q_barx 2)) `rem` 2)
      q_barx x = x*((nzc*(u+1)) `div` 31)

x_q :: Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt ->
       Data DefaultInt -> Data (Complex Float)
x_q m nzc v u = cis $ (-1)*pi*(i2f q)*(i2f m)*((i2f m)+1) / (i2f nzc)
    where q = (f2i ((q_barx 1) + 0.5))-v*(1-(2*(f2i (q_barx 2))) `rem` 2)
          q_barx x = x*((i2f (nzc*(u+1))) / 31)



-- The length  n_rs_zc of the Zadoff-Chu sequence is given by the largest
-- prime number such that _N_RS_ZC < _M_RS_SC.
-- Finds the largest prime number:
-- Function: Finds the largest prime number less than input arg m_rs_sc,
-- Application example: eval $ n_rs_zc 216
n_rs_zc :: Data DefaultInt -> Data DefaultInt
--n_rs_zc m_rs_sc = forLoop (length tolvtable)
                          0
                          (\i s ->
                              ((tolvtable)!i == m_rs_sc) ? ((primetable)!i, s))
n_rs_zc m_rs_sc = primetable!(i2n ((m_rs_sc `div` 12) - 1))

primetable :: DVector DefaultInt
primetable = vector [11, 23, 31, 47, 59, 71, 83, 89, 107, 113, 131, 139, 151,
    167, 179, 191, 199, 211, 227, 239, 251, 263, 271, 283, 293, 311, 317, 331,
    347, 359, 367, 383, 389, 401, 419, 431, 443, 449, 467, 479, 491, 503, 509,
    523, 523, 547, 563, 571, 587, 599, 607, 619, 631, 647, 659, 661, 683, 691,
    701, 719, 727, 743, 751, 761, 773, 787, 797, 811, 827, 839, 839, 863, 863,
    887, 887, 911, 919, 929, 947, 953, 971, 983, 991, 997, 1019, 1031, 1039,
    1051, 1063, 1069, 1091, 1103, 1109, 1123, 1129, 1151, 1163, 1171, 1187, 1193]

--Calculates the cyclic shift in slot n
alpha :: Data DefaultWord ->
         Data DefaultInt ->
         Data DefaultInt ->
         Data DefaultInt -> Data Float
alpha n cs csf cid = 2*pi*(i2f n_cs)/12
    where n_cs = (n_dmrs1 + n_dmrs2 + (n_prs n)) `rem` 12
          n_dmrs1 = cs_dmrs1!(i2n cs)
          n_dmrs2 = csf_dmrs2!(i2n csf)
          csf_dmrs2 = vector [0,6,3,4,2,8,10,9]
          cs_dmrs1 = vector [0,2,3,4,6,8,9,10]
          n_prs x = n_prs_all_ns cid x


-- Function: Calculates the parameter n_prs for all 20.
-- according to 3GPP 36.211 section 5.5.2.1.1
-- Application example: eval $ n_prs 16
-- Author: Emil Axelsson
n_prs_all_ns:: Data DefaultInt -> Data DefaultWord -> Data DefaultInt
n_prs_all_ns nCellId = c
```

```
  where
    delta_ss=0
    nc = 100
    n_ul_symb = 7

    c_init:: Data DefaultWord
    c_init = (i2n ((div nCellId 30)* 32 + f_ss_pusch))
       where
          f_ss_pusch=(f_ss_pucch + delta_ss)`rem` 30
          f_ss_pucch= nCellId `rem` 30

    x1_init = 1
    x2_init =  c_init

    c slot = i2n (((x1' `xor` x2') >> 21) .&. 0xFF)
      where
        (x1',x2') = forLoop (nc-21+8*n_ul_symb*slot)
                            (x1_init,x2_init)
                            (\_ -> x1_step *** x2_step)
        x1_step :: Data DefaultWord -> Data DefaultWord
        x1_step = (>>1) . combineBits xor [3,0] 31
        x2_step :: Data DefaultWord -> Data DefaultWord
        x2_step = (>>1) . combineBits xor [3,2,1,0] 31


combineBits ::
    (Data DefaultWord -> Data DefaultWord -> Data DefaultWord) ->
    [Data DefaultInt] ->
    Data DefaultInt ->
    Data DefaultWord -> Data DefaultWord
combineBits op is i reg = reg .|. (resultBit << (i2n i))
  where
    resultBit = (P.foldr1 op $ P.map (reg >>) (P.map i2n is)) .&. bit 0
```

Naive implementation of `n_prs_all_ns`:

```
n_prs_all_ns :: Data DefaultInt -> Data DefaultWord -> Data DefaultInt
n_prs_all_ns n_cellid = n_prs
    where
        m_pn = 8
        nc = 1600 --Change to 100 for faster execution time
        delta_ss = 0
        n_ul_symb = 7

        c_init :: Data DefaultInt
        c_init = (n_cellid `div` 30)*(2^5)+f_ss_pusch
            where f_ss_pucch = n_cellid `mod` 30
                  f_ss_pusch = (f_ss_pucch + delta_ss) `mod` 30

        x1_i :: Data Index -> Data DefaultInt
        x1_i = \i -> i > 0 ? (0,1)

        x2_i :: Data Index -> Data DefaultInt
        x2_i = \i -> shiftR ((bit i) .&. c_init) (i2n i)
```

```
        n_prs slot =
            i2n $ sum $ indexed m_pn (\i -> (2^i)*i2n (c (8*n_ul_symb*slot+i)))
              where
                c n = ((x1!(n+nc))+(x2!(n+nc))) `mod` 2

                x1 :: DVector DefaultInt
                x1 = forLoop (nc+m_pn+8*n_ul_symb*slot)
                             (indexed (nc+m_pn+8*n_ul_symb*slot) x1_i)
                             (\n s ->
                                indexed
                                  (length s)
                                  (\k -> (k == (n+31)) ?
                                      (((s!(n+3)+s!n) `mod` 2),s!k)))

                x2 :: DVector DefaultInt
                x2 = forLoop (nc+m_pn+8*n_ul_symb*slot)
                             (indexed (nc+m_pn+8*n_ul_symb*slot) x2_i)
                             (\n s ->
                                indexed (length s) (\k -> (k == (n+31)) ?
                                    (((s!(n+3)+s!(n+2)+s!(n+1)+s!n) `mod` 2),s!k)))
```

A.2.2 MATLAB

```
% DRS
% codegen -c -config cfg -args {72,10,5,5,16,0} Dem_RS_PUSCH

function out = Dem_RS_PUSCH(Nc, v, CS, CSF, N_cellID, Delta_ss) %#codegen
% function [DRS, alfa] = Dem_RS_PUSCH(Nc, v, CS, CSF, N_cellID, Delta_ss, slot) %#codegen
% function DRS = Dem_RS_PUSCH(Nc, u, v, n_DMRS1, CSF, n_PRS);
%
% Inputs
%   Nc      Number of sub-carriers
%   v       Base sequence number
%   n_DMRS1 Broadcased demodulation refernce signal number1
%   CSF     Cyclic shift Field number (0 to 7)
%   n_cellID Cell id number
%   Delta_ss      % Parameter configured by higher layers,
%                        %   see 36.211 section 5.5.1.3
%                            % Value range 0,...,29
%
% Output
%   DRS     Demdulation reference symbols in frequency domain
%
%
% Henrik Sahlin November 3, 2008


if ((v>0)&&(Nc<6*12))
    error('Only one base sequence if number of resource blocks less than 6')
end
if (CSF>7)
    error('Cyclic Shift Field must be less than or equal to 7')
```

```
end

%% Calculate pseudo random value to be used for "cyclic shift value"

f_ssPUCCH = mod(N_cellID, 30);
f_ssPUSCH = mod(f_ssPUCCH+Delta_ss, 30);
c_init_integer = floor(N_cellID/30)*2^5+f_ssPUSCH;


%% Group number
% See 3GPP TS 36.211 section 5.5.1.3
f_gh    = 0; % No support for group hoppning
u       = mod(f_gh + f_ssPUSCH, 30);


%%
CSF_DMRS2_table = [0,6,3,4,2,8,10,9];
CS_DMRS1_table  = [0,2,3,4,6,8,9,10];

twelves = 12*(1:100);
primelist = [11, 23, 31, 47, 59, 71, 83, 89, 107, 113, 131, 139, 151, 167, ...
    179, 191, 199, 211, 227, 239, 251, 263, 271, 283, 293, 311, 317, 331, ...
    347, 359, 367, 383, 389, 401, 419, 431, 443, 449, 467, 479, 491, 503, ...
    509, 523, 523, 547, 563, 571, 587, 599, 607, 619, 631, 647, 659, 661, ...
    683, 691, 701, 719, 727, 743, 751, 761, 773, 787, 797, 811, 827, 839, ...
    839, 863, 863, 887, 887, 911, 919, 929, 947, 953, 971, 983, 991, 997, ...
    1019, 1031, 1039, 1051, 1063, 1069, 1091, 1103, 1109, 1123, 1129, 1151, ...
    1163, 1171, 1187, 1193];

%%

assert(Nc < 1297);
coder.varsize('n', [1 1296]);
n = 0:Nc-1;

[~,idx] = min(abs(twelves-Nc));
N_ZC = primelist(idx);


q_bar = N_ZC*(u+1)/31;
q = floor(q_bar+0.5)+v*(-1)^floor(2*q_bar);

x_q = coder.nullcopy(complex(zeros(1,N_ZC)));
for m = 0:N_ZC-1
    x_q(m+1) = exp(-1j*pi*q*m*(m+1)/N_ZC);
end
ChuSequence = x_q(mod(n, N_ZC)+1);



n_DMRS1 = CS_DMRS1_table(CS+1);             % See 36.211 table 5.5.2.1.1-2
n_DMRS2 = CSF_DMRS2_table(CSF+1);             % See 36.211 table 5.5.2.1.1-1


out = coder.nullcopy(complex(zeros(20,Nc)));
```

```
for slot = 0:19
    c = PseudoRandomSequenceEML(c_init_integer, slot);
    n_PRS = sum(c((8*7*slot+1):(8*7*slot+8)).' .* (2.^(0:7) ));

    n_cs = mod(n_DMRS1 +n_DMRS2 +n_PRS, 12);    % See 36.211 section 5.5.2.1.1
    alfa = 2*pi*n_cs/12;                        % See 36.211 section 5.5.2.1.1

    DRS = exp(1j*alfa*n).*ChuSequence;
    out(slot+1,:) = DRS;
end
end


% DRS_bits
% codegen -c -config cfg -args {72,10,5,5,16,0} Dem_RS_PUSCHbits

function out = Dem_RS_PUSCHbits(Nc, v, CS, CSF, N_cellID, Delta_ss) %#codegen
% function DRS = Dem_RS_PUSCH(Nc, u, v, n_DMRS1, CSF, n_PRS);
%
% Inputs
%   Nc      Number of sub-carriers
%   v       Base sequence number
%   n_DMRS1 Broadcased demodulation refernce signal number1
%   CSF     Cyclic shift Field number (0 to 7)
%   n_cellID Cell id number
%   Delta_ss      % Parameter configured by higher layers,
%                         %   see 36.211 section 5.5.1.3
%                               % Value range 0,...,29
%
% Output
%   DRS     Demdulation reference symbols in frequency domain
%
%
% Henrik Sahlin November 3, 2008

if ((v>0)&&(Nc<6*12))
    error('Only one base sequence if number of resource blocks less than 6')
end
if (CSF>7)
    error('Cyclic Shift Field must be less than or equal to 7')
end

%% Calculate pseudo random value to be used for "cyclic shift value"

f_ssPUCCH = mod(N_cellID, 30);
f_ssPUSCH = mod(f_ssPUCCH+Delta_ss, 30);


%% Group number
% See 3GPP TS 36.211 section 5.5.1.3
f_gh    = 0; % No support for group hoppning
u       = mod(f_gh + f_ssPUSCH, 30);


%%
CSF_DMRS2_table = [0,6,3,4,2,8,10,9];
```

```
CS_DMRS1_table  = [0,2,3,4,6,8,9,10];

twelves = 12*(1:100);
primelist = [11, 23, 31, 47, 59, 71, 83, 89, 107, 113, 131, 139, 151, ...
    167, 179, 191, 199, 211, 227, 239, 251, 263, 271, 283, 293, 311, ...
    317, 331, 347, 359, 367, 383, 389, 401, 419, 431, 443, 449, 467, ...
    479, 491, 503, 509, 523, 523, 547, 563, 571, 587, 599, 607, 619, ...
    631, 647, 659, 661, 683, 691, 701, 719, 727, 743, 751, 761, 773, ...
    787, 797, 811, 827, 839, 839, 863, 863, 887, 887, 911, 919, 929, ...
    947, 953, 971, 983, 991, 997, 1019, 1031, 1039, 1051, 1063, 1069, ...
    1091, 1103, 1109, 1123, 1129, 1151, 1163, 1171, 1187, 1193];

%%
assert(Nc < 1297);
coder.varsize('n', [1 1296]);
n = 0:Nc-1;

[~,idx] = min(abs(twelves-Nc));
N_ZC = primelist(idx);


q_bar = N_ZC*(u+1)/31;
q = floor(q_bar+0.5)+v*(-1)^floor(2*q_bar);

x_q = coder.nullcopy(complex(zeros(1,N_ZC)));
for m = 0:N_ZC-1
    x_q(m+1) = exp(-1j*pi*q*m*(m+1)/N_ZC);
end
ChuSequence = x_q(mod(n, N_ZC)+1);


n_DMRS1 = CS_DMRS1_table(CS+1);             % See 36.211 table 5.5.2.1.1-2
n_DMRS2 = CSF_DMRS2_table(CSF+1);             % See 36.211 table 5.5.2.1.1-1

out = coder.nullcopy(complex(zeros(20,Nc)));

for slot = 0:19
    n_PRS = double(n_prs_all_ns(N_cellID,slot));

    n_cs = mod(n_DMRS1 +n_DMRS2 +n_PRS, 12);    % See 36.211 section 5.5.2.1.1
    alfa = 2*pi*n_cs/12;                        % See 36.211 section 5.5.2.1.1

    DRS = exp(1j*alfa*n).*ChuSequence;

    out(slot+1,:) = DRS;
end


% Helper Function to DRS - calculating a pseudo-random bit sequence
function c = PseudoRandomSequenceEML(c_init_integer,slot) %#codegen

% Pseudo random sequence according to 3GPP TS 36.211, section 7.2

%c_init_bin = dec2bin(c_init_integer);   % Convert to binary 'char'
%c_init = str2num(c_init_bin(:));        %#ok<ST2NM> % Convert to vector with integers
```

```
% dec2bin and str2num are not supported by EML for code generation, let's
% make something which is (bottom).

N_c = 100;
M_PN = 8;

c_init = int2binlist(c_init_integer);
x1 = zeros(N_c+M_PN+8*7*slot+31,1);
x1(1) = 1;
x2 = zeros(N_c+M_PN+8*7*slot+31,1);
x2(1:length(c_init)) = flipud(c_init(:));
% Such that c_init_integer = Sum(x2(k) * 2^k)

for n = 0:(N_c+M_PN-1+8*7*slot)
    x1(n+31+1) = mod(x1(n+3+1) + x1(n+1), 2);
    x2(n+31+1) = mod(x2(n+3+1) + x2(n+2+1) + x2(n+1+1) + x2(n+1), 2);
end
n = 0:(M_PN-1+8*7*slot);
c = mod(x1(n+N_c+1) + x2(n+N_c+1),2);

end

function out = int2binlist(input)
    np = nextpow2(input);
    input = uint32(input);
    assert(np < 8);
    out = zeros(np+1, 1);
    for i = 1:np+1
       out(np+2-i) = bitshift(bitand(input, 2^(i-1)), -np);
    end
end


% Helper Function to DRS_bits - calculating a pseudo-random bit sequence
function out = n_prs_all_ns(nCellId,slot) %#codegen

    delta_ss = 0;
    f_ss_pucch = mod(nCellId, 30);
    f_ss_pusch = mod((f_ss_pucch + delta_ss), 30);
    c_init = idivide(int32(nCellId), int32(30)) * 32 + f_ss_pusch;
    x1_init = 1;
    x2_init = uint32(c_init);

    x1_ = uint32(0);
    x2_ = uint32(0);

    x1 = uint32(x1_init);
    x2 = uint32(x2_init);
    for k = 1:(79+8*7*slot) %1579

       list = bitshift(x1, -[0,3]);
       resultBit = list(1);

       for i = list(2:end)
           resultBit = bitxor(i, resultBit);
       end
```

```
        resultBit = bitand(resultBit, 1);
        x1 = bitor(x1, bitshift(resultBit, 31));
        x1_ = bitshift(x1, -1);
        x1 = x1_;

        list = bitshift(x2, -[0,1,2,3]);
        resultBit = list(1);

        for i = list(2:end)
            resultBit = bitxor(i, resultBit);
        end

        resultBit = bitand(resultBit, 1);
        x2 = bitor(x2, bitshift(resultBit, 31));
        x2_ = bitshift(x2, -1);
        x2 = x2_;
    end

    out = bitand(bitshift(bitxor(x1_, x2_),-21),255);

end
```

A.3 ChEst

A.3.1 Feldspar

```
import qualified Prelude
import Feldspar
import Feldspar.Compiler
import Feldspar.Vector
import Fft
import PlatformAnsiC
import Feldspar.Matrix

_D :: Data DefaultWord
_D = 2

_L :: Data DefaultWord -> Data DefaultWord
_L n = ceiling $ i2f n / 10

nextpow2 :: Data DefaultWord -> Data DefaultWord
nextpow2 x = ceiling $ logBase 2 $ i2f x

xt :: DVector (Complex Float) ->
      Data DefaultWord -> DVector (Complex Float)
xt x n = complex root 0 *** ifft x
  where
    root = sqrt $ i2f $ _D * n

xt_window :: DVector (Complex Float) ->
```

```
            Data DefaultWord -> DVector (Complex Float)
xt_window x n = take winlen x ++ replicate (2^(nextpow2 (_D * n)) - winlen) 0
  where
    winlen = _L n * _D + 1

xf :: DVector (Complex Float) -> Data DefaultWord -> DVector (Complex Float)
xf x n = (1/complex root 0) *** fft (xt_window (xt x n) n)
  where
    root = sqrt $ i2f $ _D * n

chest_DFT :: DVector (Complex Float) ->
             Data DefaultWord ->
             (DVector (Complex Float), DVector (Complex Float))
chest_DFT x n = (xt x n, xf x n)

chest_DFT_wrap :: Data [Complex Float] ->
                  (DVector (Complex Float), DVector (Complex Float))
chest_DFT_wrap x =
    (xt (unfreezeVector' 1200 x) 2048, xf (unfreezeVector' 1200 x) 2048)
```

A.3.2 MATLAB

```
% ChEst - compiled for 1200 sub-carriers
% codegen -c -config cfg -args {complex(zeros(1,64))} chest_DFT600

function [Xf, xt] = chest_DFT1200(X) %#codegen

N = 1200;
D = 2;
L = ceil(N/10);

%% Time domain
xt      = sqrt(D*N)*ifft(X);


%% Window


xt_window            = complex(zeros(1,2^nextpow2(D*N)));
xt_window(1:(L*D+1)) = xt(1:(L*D+1));


%% Frequency domain again
Xf      = 1/sqrt(D*N)*fft(xt_window);
```

A.4 MMSE

A.4.1 Feldspar

Feldspar

```
import qualified Prelude
import Feldspar
import Feldspar.Compiler
import Feldspar.Matrix
import Feldspar.Vector
import PlatformAnsiC
import Feldspar.Compiler.Backend.C.Options


{-
  MMSE based equalization including some poor functions for matrix inverse
-}

--
-- inverseMatrix using vector library
--

inverseMatrix :: Matrix (Complex Float) -> Matrix (Complex Float)
inverseMatrix m = augmentedPart $ rrowEchelon $ augmentMatrix m $ idMatrix $ length m

-- Take right part of augmented matrix
augmentedPart :: Matrix (Complex Float) -> Matrix (Complex Float)
augmentedPart m = map (drop (length m)) m

-- Put matrix m2 to the right of matrix m1
augmentMatrix :: Matrix (Complex Float) ->
                 Matrix (Complex Float) -> Matrix (Complex Float)
augmentMatrix = zipWith (++)

-- nxn identity matrix
idMatrix :: Data DefaultWord -> Matrix (Complex Float)
idMatrix n = indexedMat n n (\i j -> complex (i2f ((b2i (i == j))::Data DefaultInt)) 0)

-- Apply Gauss-Jordan elimination to obtain reduced row echelon form
rrowEchelon :: Matrix (Complex Float) -> Matrix (Complex Float)
rrowEchelon m = forLoop (length m) m fun
  where
    fun i state = ((state!piv)!i /= 0) ? (subaru, state)
      where
            piv = pivot i state
            swapped = swap state i piv
            divided = divRowInMatrix swapped i (swapped!i!i)
            subaru = indexed (length m) allButI
            allButI n = (n /= i) ? (subRow (divided!n) (divided!i) i, divided!n)

-- Find pivot in column i, starting in row i
pivot :: Data DefaultWord -> Matrix (Complex Float) -> Data DefaultWord
```

```
pivot i m = forLoop (length m - 1) i piv
  where
    piv k s = (magnitude (m!(k+1)!i) > magnitude (m!s!i)) ? (k+1, s)

-- Swap rows i and j
swap :: Matrix (Complex Float) ->
        Data DefaultWord ->
        Data DefaultWord -> Matrix (Complex Float)
swap m i j = permute (\_ k -> (k == i) ? (j,((k == j) ? (i,k)))) m

-- Divide all elements in row k with x
divRowInMatrix :: Matrix (Complex Float) ->
                  Data DefaultWord ->
                  Data (Complex Float)-> Matrix (Complex Float)
divRowInMatrix m k x =
  indexed (length m) $ \i -> (i == k) ? (map (/x) (m!i), m!i)

-- Subtract x .* is from v
subRow :: DVector (Complex Float) ->
          DVector (Complex Float) ->
          Data DefaultWord -> DVector (Complex Float)
subRow v is j = v .- (is *** (v!j))

--
-- inverseMatrix using core matrices
--

inverseMatrix2 :: Matrix (Complex Float) -> Matrix (Complex Float)
inverseMatrix2 m = augmentedPart $ rrowEchelon_core $
                      augmentMatrix m $ idMatrix $ length m

-- Gauss-Jordan elimination with core matrix in forLoop state
rrowEchelon_core :: Matrix (Complex Float) -> Matrix (Complex Float)
rrowEchelon_core m = unfreezeMatrix $ forLoop rows (freezeMatrix m) fun
  where
    rows = length m
    cols = length $ head m
    fun i state = ((state!piv)!i /= 0) ? (subaru, state)
      where
        subaru = parallel rows allButI
        swapped = swap_core state i piv
        divided = divRowInMatrix_core swapped i ((swapped!i)!i) cols
        piv = pivot_core i state rows
        allButI n = (n /= i) ? (subRow_core (divided!n) (divided!i) i cols, divided!n)

pivot_core :: Data DefaultWord ->
              Data [[Complex Float]] ->
              Data DefaultWord -> Data DefaultWord
pivot_core i m len = forLoop (len-1) i piv
  where piv k s = (magnitude (m!(k+1)!i) > magnitude (m!s!i)) ? (k+1, s)

-- Swap rows in core matrix
swap_core :: Data [[Complex Float]] ->
        Data DefaultWord ->
        Data DefaultWord -> Data [[Complex Float]]
swap_core m i j = setIx (setIx m i (m!j)) j (m!i)
```

```
divRowInMatrix_core :: Data [[Complex Float]] ->
                   Data DefaultWord ->
                   Data (Complex Float) ->
                   Data DefaultWord -> Data [[Complex Float]]
divRowInMatrix_core m k x len = setIx m k newRow
  where
        newRow = parallel len (\i -> (m!k)!i / x)

subRow_core :: Data [Complex Float] ->
               Data [Complex Float] ->
               Data DefaultWord ->
               Data DefaultWord -> Data [Complex Float]
subRow_core v is j len = parallel len (\i ->  v!i -  multiplied!i)
  where multiplied = parallel len (\i -> is!i * v!j)

-- MMSE based equalization and antenna combining
-- Equal amount of Rx and Tx antennas
-- w = h' * (h*h'+c)^-1 where ' is the complex conjugate transpose
mmse_eq1 :: Vector (Matrix (Complex Float)) ->
            Matrix (Complex Float) ->
            (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq1 h c = (indexed n w, indexed n hp)
  where
    w k = fnutt (h!k) *** inverseMatrix ((h!k) *** fnutt (h!k) .+ c)
    hp k = w k *** (h!k)
    n = length h

un3 :: Type a => Data [[[a]]] -> Vector (Matrix a)
un3 = map (map (unfreezeVector' 8)) . map (unfreezeVector' 8) . unfreezeVector' 1200

un2 :: Type a => Data [[a]] -> Matrix a
un2 = map (unfreezeVector' 8) . unfreezeVector' 8

mmse_eq1_wrap :: Data [[[Complex Float]]] ->
                 Data [[Complex Float]] ->
                    (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq1_wrap h c = mmse_eq1 (un3 h) (un2 c)

-- MMSE based equalization and antenna combining
-- Equal amount of Rx and Tx antennas
-- w = h' * (h*h'+c)^-1 where ' is the complex conjugate transpose
mmse_eq1_core :: Vector (Matrix (Complex Float)) ->
            Matrix (Complex Float) ->
            (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq1_core h c = (indexed n w, indexed n hp)
  where
    w k = fnutt (h!k) *** inverseMatrix2 ((h!k) *** fnutt (h!k) .+ c)
    hp k = w k *** (h!k)
    n = length h

mmse_eq1_core_wrap :: Data [[[Complex Float]]] ->
            Data [[Complex Float]] ->
            (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq1_core_wrap h c = mmse_eq1_core (un3 h) (un2 c)
```

```
-- Less Tx antennas than Rx antennas
-- w = (inv(h' * inv(c)*h+(id n_tx))*h')*inv(C)
mmse_eq2 :: Vector (Matrix (Complex Float)) ->
            Matrix (Complex Float) ->
            (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq2 h c = (indexed n w, indexed n hp)
  where
    w k = (inverseMatrix (fnutt (h!k)
          *** inverseMatrix c
          *** (h!k) .+ (idMatrix n_tx))
          *** fnutt (h!k))
          *** inverseMatrix c
    hp k = w k *** (h!k)
    n_tx = length $ head $ head h
    n = length h

-- MMSE EQ 2 with core matrix inversion
mmse_eq2_core :: Vector (Matrix (Complex Float)) ->
            Matrix (Complex Float) ->
            (Vector (Matrix (Complex Float)), Vector (Matrix (Complex Float)))
mmse_eq2_core h c = (indexed n w, indexed n hp)
  where
    w k = (inverseMatrix2 (fnutt (h!k)
          *** inverseMatrix2 c
          *** (h!k) .+ (idMatrix n_tx))
          *** fnutt (h!k))
          *** inverseMatrix2 c
    hp k = w k *** (h!k)
    n_tx = length $ head $ head h
    n = length h

-- Complex conjugate transpose
fnutt :: Matrix (Complex Float) -> Matrix (Complex Float)
fnutt = map (map conjugate) . transpose
```

A.4.2 MATLAB

```
% MMSE1
% codegen -c -config cfg -args {complex(zeros(8,8,1200)), zeros(8,8)} MMSE_EQ1

function [W, H_post] = MMSE_EQ1(H, C) %#codegen

% MMSE based equalization and atenna combining
%
% This formulation if preferred if
%   equal amount of Rx and Tx antennas
% H' * inv(H*H'+C)
[N_Rx, N_Tx, N] = size(H);

W        = coder.nullcopy(complex(zeros(N_Tx, N_Rx, N)));
H_post   = coder.nullcopy(complex(zeros(N_Tx, N_Tx, N)));
for k=1:N
    W(:,:,k)        = H(:,:,k)'/(H(:,:,k)*H(:,:,k)'+C);
```

```
    H_post(:,:,k)   = W(:,:,k) * H(:,:,k);
end


% MMSE1_opt
% codegen -c -config cfg -args {complex(zeros(8,8,1200)), zeros(8,8)} MMSE_EQ1_opt

function [W, H_post] = MMSE_EQ1_opt(H, C) %#codegen

% MMSE based equalization and atenna combining
%
% This formulation if preferred if
%   equal amount of Rx and Tx antennas
% H' * inv(H*H'+C)
[N_Rx, N_Tx, N] = size(H);

W        = coder.nullcopy(complex(zeros(N_Tx, N_Rx, N)));
H_post   = coder.nullcopy(complex(zeros(N_Tx, N_Tx, N)));
for k=1:N
    Hk              = H(:,:,k);
    Wk              = Hk'/(Hk*Hk'+C);
    H_post(:,:,k)   = Wk * Hk;
    W(:,:,k)        = Wk;
end


% MMSE2
% codegen -c -config cfg -args {complex(zeros(8,2,1200)), zeros(8,8)} MMSE_EQ2

function [W, H_post] = MMSE_EQ2(H, C) %#codegen

% MMSE based equalization and atenna combining
%
% This formulation if preferred if
%  less Tx antennas than Rx antennas
%inv(H(:,:,k)'*inv(C)*H(:,:,k)+eye(N_Tx))*H(:,:,k)'*inv(C);

[N_Rx, N_Tx, N] = size(H);

W        = coder.nullcopy(complex(zeros(N_Tx, N_Rx, N)));
H_post   = coder.nullcopy(complex(zeros(N_Tx, N_Tx, N)));
for k=1:N
    W(:,:,k)        = (H(:,:,k)'/C*H(:,:,k)+eye(N_Tx))\H(:,:,k)'/C;
    H_post(:,:,k)   = W(:,:,k) * H(:,:,k);
end


% MMSE2_opt
% codegen -c -config cfg -args {complex(zeros(8,2,1200)), zeros(8,8)} MMSE_EQ2_opt

function [W, H_post] = MMSE_EQ2_opt(H, C) %#codegen

% MMSE based equalization and atenna combining
%
% This formulation if preferred if
%  less Tx antennas than Rx antennas
```

```
%inv(H(:,:,k)'*inv(C)*H(:,:,k)+eye(N_Tx))*H(:,:,k)'*inv(C);

[N_Rx, N_Tx, N] = size(H);

W        = coder.nullcopy(complex(zeros(N_Tx, N_Rx, N)));
H_post   = coder.nullcopy(complex(zeros(N_Tx, N_Tx, N)));
for k=1:N
    Hk = H(:,:,k);
    tmp = (Hk'/C)*Hk;
    % Add eye(N_Tx) without forming the identity matrix.
    for j = 1:N_Tx
       tmp(j,j) = tmp(j,j) + 1;
    end
    Wk = (tmp \ Hk') / C;
    W(:,:,k) = Wk;
    H_post(:,:,k) = Wk * Hk;
end
```

A.5 MATLAB Coder Configuration

```
Description: 'class EmbeddedCodeConfig:
              C code generation Embedded Coder configuration objects.'

    Name: 'EmbeddedCodeConfig'

-------------------------------- Report --------------------------------

                GenerateReport: true
                  LaunchReport: false

--------------------------- Code Generation --------------------------

            FilePartitionMethod: 'MapMFileToCFile'
                    GenCodeOnly: true
               GenerateMakefile: true
                    MakeCommand: 'make_rtw'
              MultiInstanceCode: false
                     OutputType: 'EXE'
             PostCodeGenCommand: ''
                     TargetLang: 'C'
              TemplateMakefile: 'default_tmf'

----------------------- Language And Semantics -----------------------

         ConstantFoldingTimeout: 10000
        DynamicMemoryAllocation: 'Off'
           EnableVariableSizing: true
          InitFltsAndDblsToZero: true
              PurelyIntegerCode: false
      SaturateOnIntegerOverflow: false
                SupportNonFinite: false
```

```
          TargetFunctionLibrary: 'C99 (ISO)'

---------------- Function Inlining and Stack Allocation ---------------

               InlineStackLimit: 4000000
                InlineThreshold: 700
             InlineThresholdMax: 800
                  StackUsageMax: 200000

----------------------------- Optimizations ---------------------------

    CCompilerCustomOptimizations: ''
          CCompilerOptimization: 'Off'
              ConvertIfToSwitch: false
                   EnableMemcpy: true
                 MemcpyThreshold: 64

------------------------------- Comments ------------------------------

               GenerateComments: true
                  MATLABFcnDesc: false
           MATLABSourceComments: false
                        Verbose: false

------------------------------ Custom Code ----------------------------

               CustomHeaderCode: ''
                  CustomInclude: ''
              CustomInitializer: ''
                  CustomLibrary: ''
                   CustomSource: ''
               CustomSourceCode: ''
               CustomTerminator: ''
              ReservedNameArray: ''

------------------------------ Code Style -----------------------------

        CustomSymbolStrEMXArray: 'emxArray_$M$N'
             CustomSymbolStrFcn: '$M$N'
           CustomSymbolStrField: '$M$N'
       CustomSymbolStrGlobalVar: '$M$N'
           CustomSymbolStrMacro: '$M$N'
          CustomSymbolStrTmpVar: '$M$N'
            CustomSymbolStrType: '$M$N'
            IncludeTerminateFcn: true
                    MaxIdLength: 31
              ParenthesesLevel: 'Nominal'
       PreserveExternInFcnDecls: true

------------------------------ Hardware -------------------------------

         HardwareImplementation: [1x1 coder.HardwareImplementation]
```

Appendix B

Appendix: Results - Execution Time

B.1 PC

Test Program	F #1	F #2	F #3	M #1	M #2	M #3
BinarySearch	12	10	15	1	4	4
BitRev	1380	2832	6141	540	1340	2521
Bubblesort1	102	1162	8715	180	1082	3373
Bubblesort2	582	3703	29287	-	-	-
Bubblesort opt	-	-	-	200	1078	3510
DotRev	101	213	426	222	453	865
DotRev opt	-	-	-	70	109	220
Duplicate	130	251	423	0	67	197
Duplicate2	15500	73227	293103	-	-	-
IdMatrix	119	436	1924	25	102	1062
SliceMatrix	536	2090	8250	395	1520	7009
RevRev	566	1096	2275	570	1171	2236
SqAvg	1759	3491	6879	2106	4264	8537
SqAvg opt	-	-	-	1729	3401	6941
TwoFir	155	344	1031	6	16	49
TwoFir wrap	-	-	368	-	-	-
DRS	-	-	-	2566	3468	4434
DRS opt	6455	9128	11942	1801	2693	3873
ChEst	5192	5017	86964	422	485	703
ChEst wrap	-	-	71842	-	-	-
MMSE EQ1	223	1876	7849	1	45	264
MMSE EQ1 core	217	1896	7514	-	-	-
MMSE EQ1 wrap	-	-	2888	-	-	-
MMSE EQ1 core wrap	-	-	2853	-	-	-
MMSE EQ1 opt	-	-	-	1	38	232
MMSE EQ2	-	-	-	1	0	0
MMSE EQ2 core	28	253	716	-	-	-
MMSE EQ2 opt	-	-	-	0	0	1

Table B.1: Number of sampled CPU cycles for each run. F is Feldspar and M is MATLAB. - means that the test program was not implemented or run.

B.2 TI C6670 Simulator

Test Program	F #1	F #2	F #3	M #1	M #2	M #3
BinarySearch	1390	9175	9865	769	876	981
BitRev	1120	27500	66222	1081	2341	4749
Bubblesort1	21563	1330177	10424269	9279	37420	216248
Bubblesort2	42280	1389005	11036332	-	-	-
Bubblesort opt	-	-	-	3860	14103	75383
DotRev	9112	16735	32936	24265	48521	96957
DotRev opt	-	-	-	3493	6841	13292
Duplicate	19744	37579	71831	15827	31199	63440
Duplicate2	22959	3232447	3219960	-	-	-
IdMatrix	8374	33997	136394	1438	6461	26148
SliceMatrix	4073	14036	51745	5202	17253	60622
RevRev	9971	17656	37592	17895	36719	73315
SqAvg	6711	13867	27133	25530	47736	93311
SqAvg opt	-	-	-	25764	48291	94436
TwoFir	272170	3220941	11163382	196633	809926	1636110
DRS	-	-	-	3678565	3877449	4102941
DRS opt	3834717	4993185	6144403	1189953	1393188	1590246
ChEst	1601114	1960925	2778733	25485	31708	45730
MMSE EQ1	5848081	40402456	308317922	124868	787355	5185359
MMSE EQ1 core	5383219	33892362	250526324	-	-	-
MMSE EQ1 opt	-	-	-	128536	848423	4135210
MMSE EQ2	-	-	-	104087	515888	1115753
MMSE EQ2 core	1275381	8058645	27297640	-	-	-
MMSE EQ2 opt	-	-	-	103626	531380	1039320

Table B.2: Number of CPU cycles for each run. F is Feldspar and M is MATLAB. - means that the test program was not implemented or run.

Appendix C

Appendix: Survey

C.1 Questions

(0a) MATLAB experience.
No Experience (1-10) Expert

(0b) Functional programming experience.
No Experience (1-10) Expert

(0c) Seen Feldspar?
(Yes/No)

1a

Consider the following MATLAB function f

```
function out = f(v)

out = (v*v.')/length(v);

end
```

How easy is it to understand what f does?
Easy (1-10) Hard

1b

Now consider the Feldspar function f

```
f :: DVector Float -> DVector Float
```

```
f vec = zipWith g vec (tail vec)
   where g x y = 0.5*x + 0.5*y
```

How easy is it to understand what f does?
Easy (1-10) Hard

2a

Consider the following Feldspar functions implementing two FIR filters in series (think of k1 and k2 as arbitrary coefficient vectors).

```
filter1 :: DVector Float -> DVector Float
filter1 = convolution k1

filter2 :: DVector Float -> DVector Float
filter2 = convolution k2

filter_chain :: DVector Float -> DVector Float
filter_chain = filter2 . filter1
```

Now look at the MATLAB implementation of the same filter (think of k1 and k2 as arbitrary coefficient vectors).

```
function v = filter_chain(v, k1, k2)

v = conv(conv(v,k1),k2);

end
```

Which implementation do you find more intuitive?
Feldspar (1-10) MATLAB

2b

The function cumxor takes a start value s and a vector v of unsigned integers. It does bitwise XOR between s and the first element of v, and then does a new XOR between the result and the second element of v, and so on until the end of v. The final result is then returned.

Feldspar implementation:

```
cumxor :: Data DefaultWord -> DVector DefaultWord -> Data DefaultWord
cumxor = fold xor
```

MATLAB implementation:

```
function s = cumxor(s, v)
  for i = 1:length(v)
     s = bitxor(v(i), s);
  end
end
```

Which implementation do you find more intuitive?
Feldspar (1-10) MATLAB

3

The following function computes the square root of the i:th element of the vector v.

Feldspar implementation:

```
fun :: DVector Float -> Data Index-> Data Float
fun v i = sqrt (v!i)
```

MATLAB implementation:

```
function out = fun(v, i)
    out = sqrt(v(i));
end
```

Change the functions so that they do square root on all elements in v.

(3a) How easy was it for the Feldspar implementation?
Easy (1-10) Hard

(3b) How easy was it for the MATLAB implementation?
Easy (1-10) Hard

C.2 Answers

Participant	0a	0b	0c	1a	1b	2a	2b	3a	3b
1	7	2	No	2	8	10	9	8	1
2	9	2	No	1	9	8	8	4	1
3	9	2	No	1	8	8	8	5	2
4	8	2	Yes	2	8	7	9	8	2
5	3	10	Yes	7	2	2	4	1	3
6	3	2	Yes	3	7	3	8	1	1
7	2	10	No	10	2	1	1	2	10
8	8	1	No	1	9	9	9	10	1
9	5	2	Yes	1	6	5	3	2	2

Table C.1: Answers to the survey.

Bibliography

[1] Axelsson E, Claessen K, Dévai G, Horváth Z, Keijzer K, Lyckegård B, Persson A, Sheeran M, Svenningsson J and Vajda A. *Feldspar: A Domain Specific Language for Digital Signal Processing Algorithms.* Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, 2010.

[2] Axelsson E, Persson A, Sheeran M, Svenningsson J and Dévai G. *A Tutorial on Programming in Feldspar.* `http://feldspar.inf.elte.hu/feldspar/documents/FeldsparTutorial.pdf`, 2011.

[3] MathWorks. *http://www.mathworks.com/products/matlab/*, May 2011.

[4] *MATLAB User's Guide (MATLAB Coder).* `http://www.mathworks.com/help/toolbox/coder/index.html`. Version: 2011a.

[5] MathWorks. *MATLAB Language Subset for Code Generation - MATLAB Coder.* `http://www.mathworks.com/products/matlab-coder/description2.html`, May 2011.

[6] *Single Assignment C – Functional Array Programming for High-Performance Computing.* `http://www.sac-home.org/`, January 2011.

[7] Vikström A. *A Study of Automatic Translation of MATLAB Code to C Code using Software from the MathWorks.* Master's thesis, Luleå University of Technology, 2009.

[8] Müllegger M. *Evaluation of Compilers for MATLAB- to C-Code Translation.* Master's thesis, Halmstad University, 2008.

[9] *The Glasgow Haskell Compiler.* `http://www.haskell.org/ghc/`, May 2011.

[10] *The Haskell Platform.* `http://hackage.haskell.org/platform/`, May 2011.

[11] Hutton G. *Programming in Haskell.* Cambridge University Press, 2007.

[12] Thompson S. *Haskell: The Craft of Functional Programming.* Icss Series. Pearson Education, Limited, 2006.

[13] Coutts D, Leshchinskiy R and Stewart D. *Stream Fusion From Lists to Streams to Nothing at All.* ICFP'07 Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, 2007.

[14] *User's Guide to the Feldspar Compiler.* `http://feldspar.inf.elte.hu/feldspar/documents/FeldsparCompilerDocumentation.html`, 2011.

[15] *MATLAB User's Guide (Fixed-Point Toolbox).* `http://www.mathworks.com/help/toolbox/fixedpoint/`. Version: 2011a.

[16] MathWorks. *MathWorks - MATLAB and Simulink for Technical Computing.* `http://www.mathworks.com/`, May 2011.

[17] Moler C. *The Origins of MATLAB.* `http://www.mathworks.com/company/newsletters/news\discretionary{-}{}{}notes/clevescorner/dec04.html`, May 2011.

[18] MathWorks. *Mathtools.net : MATLAB*. http://www.mathtools.net/MATLAB/index.html, May 2011.

[19] *MATLAB User's Guide (Object-Oriented Programming)*. http://www.mathworks.com/help/techdoc/matlab_oop/ug_intropage.html. Version: 2011a.

[20] *MATLAB User's Guide (Techniques for Improving Performance)*. http://www.mathworks.com/help/techdoc/matlab_prog/f8-784135.html#br8fs0d-1. Version: 2011a.

[21] *MATLAB User's Guide (Assert)*. http://www.mathworks.com/help/techdoc/ref/assert.html. Version: 2011a.

[22] *MATLAB User's Guide (Parallel Computing Toolbox)*. http://www.mathworks.com/help/toolbox/distcomp/. Version: 2011a.

[23] *MATLAB User's Guide (Parallel Computing Toolbox - parfor)*. http://www.mathworks.com/help/toolbox/distcomp/parfor.html. Version: 2011a.

[24] *3GPP 36.211 Technical Specification*. www.quintillion.co.jp/3GPP/Specs/36211-910.pdf, 2010.

[25] Press W. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 2007.

[26] Cheney W and Kincaid D. *Linear Algebra: Theory and Applications*. Jones & Bartlett Publishers, Incorporated, 2011.

[27] *MATLAB User's Guide (mldivide, mrdivide)*. http://www.mathworks.com/help/techdoc/ref/mldivide.html. Version: 2011a.

[28] Texas Instruments. *C6000 High Performance Multicore DSP - TMS320C66x DSP - TMS320C6670 - TI.com*. http://focus.ti.com/docs/prod/folders/print/tms320c6670.html, May 2011.

[29] Aggarwal K, Singh Y and Chhabra J. *An Integrated Measure of Software Maintainability*. Reliability and Maintainability Symposium, 2002. 235–241.

[30] Pierce B. *Types and programming languages*. MIT Press, 2002.

[31] Spinellis D. *Code documentation*. IEEE Software, 2010. 27(4):18–19.

[32] Venners B and Thomas D. *Orthogonality and the DRY Principle*. http://www.artima.com/intv/dry.html, May 2011.

[33] Goodliffe P. *Code craft: the practice of writing excellent code*. No Starch Press Series. No Starch Press, 2007.

[34] Buse R and Weimer W. *Learning a Metric for Code Readability*. IEEE Transactions on Software Engineering, 2010. 36(4):546–558.

[35] Wallace D, Ippolito L and Cuthill B. *Reference Information for the Software Verification & Validation Process*. Diane Pub Co, 1996.

[36] Zoffmann G, Gingerl M, Reumann C and Sonneck G. *A Classification Scheme for Software Verification Tools with Regard to RTCA/DO-178B*. SAFECOMP 2001, 2001. Chapter 6.

[37] D'Silva V, Kroening D and Weissenbacher G. *A Survey of Automated Techniques for Formal Software Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008. 27(7). Section II - E.

[38] Abran A, Moore J, Bourque P and Dupuis R. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004. Chapter 5.

[39] Marciniak J. *Encyclopedia of software engineering*, 970–978. J. Wiley, 1994. D. Hamlet, Random Testing.

[40] *MATLAB User's Guide (Concatenating Objects of Different Classes).* `http://www.mathworks.com/help/techdoc/matlab_oop/bsfenzt.html`. Version: 2011a.

[41] *MATLAB User's Guide (Debugging Process and Features).* `http://www.mathworks.com/help/techdoc/matlab_env/brqxeeu-175.html`. Version: 2011a.

[42] *The GHCi Debugger.* `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci-debugger.html`, May 2011.

[43] Marlow S. *Haddock: A Haskell Documentation Tool.* `http://www.haskell.org/haddock`, May 2011.

[44] van Heesch D. *Doxygen.* `http://www.doxygen.org`, May 2011.

[45] *Gnuplot - HaskellWiki.* `http://www.haskell.org/haskellwiki/Gnuplot`, May 2011.

[46] Claessen K and Hughes J. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.* ACM SIGPLAN Notices, 2000. 35(9).

[47] *QuickCheck Manual.* `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`.