MATFORCE: SUPPORTING RAPID ALGORITHM DEVELOPMENT BY AUTOMATED TRANSLATION OF MATLAB PROTOTYPES INTO C++

Levente Hunyadi
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
1111 Budapest, Magyar tudósok körútja 2., Hungary
e-mail: hunyadi@aut.bme.hu

ABSTRACT
MatLab is an essential tool in high-productivity development of applications that involve much scientific computation. Problems can be presented in a familiar mathematical formalism and the simple yet extensive visualization capabilities support rapid algorithm and model prototyping. Nonetheless, for the sake of efficiency and homogeneity with other parts of the code, it is often necessary to convert MatLab code into C or C++, which is a tedious and error-prone task if performed manually. The author presents a tool named MatForce that automatically converts MatLab scripts into C++ code, producing human-readable, extensible C++ sources that can subsequently be fitted to the needs of the encapsulating application.

KEYWORDS
programming tools and languages, developing computation-intensive algorithms, source-to-source compilation, automatic type inference

1 Introduction

MatLab is the de-facto language of technical computing, it allows programmers to write algorithms in familiar mathematical notation as well as visualize their results in a straightforward manner. On one hand, MatLab has built-in support for matrix operations and a myriad of mathematical functions ranging from trigonometric functions to fast Fourier transformation. On the other hand, data can be displayed graphically and interactive tools allow manipulating graphs to achieve results that reveal the most information. Both of these aspects contribute to ease of use and quick initial discovery of potential pitfalls. However, beyond the initial prototyping phase, one is often faced with the need to integrate the developed algorithms into an existing application, often with subtle changes.

Performance is another strong motivation behind code translation. As MatLab is an interpreted language, its raw speed is vastly inferior to such compiled programming languages as C or C++.[1] Albeit MatLab uses highly optimized matrix algebra and mathematical function libraries, extra processing associated with executing individual MatLab statements incurs a relatively large processor footprint, which is apparent especially in the case of programs that contain many control flow statements such as conditionals and loops.

In order to ease integration and lessen computation overhead, MatLab ships with a compiler that automatically converts MatLab code (so-called *m-files*) into C or C++ shared libraries. While exhibits a speed gain as compared to directly interpreted code, this approach has several limitations:

- *Loss of type safety.* The libraries expose their interface through C header files that use intricate MatLab-style arrays, which are not intuitive when used in the host application.[2] In general, very little attempt is made to use native C types. As MatLab itself is a *dynamically typed* language[3], there is little provision for type safety when invoking compiled MatLab functions.
- *External dependence.* The shared libraries rely on an external, relatively heavy-weight library called the MatLab Component Runtime (MCR), which must also be installed on the end-user's computer.
- *Loss of extensibility.* Albeit C or C++ header files are generated during the compilation phase, function definitions in m-files are translated into binary rather than source code. As a result, compiled files are no longer editable to be extended with additional functionality that may not have been possible at the MatLab level.

In order to remedy the outlined limitations, the author describes *MatForce* [4], an open-source tool comprising of a compiler and a utility library, which translates MatLab code into human-readable C++ source code. The *compiler*[4] features a simple yet powerful type inference algorithm that uses primitive as well as compound types in compiled code. The resultant C++ code is dependent only on

[1] Just-in-time compilation can, in part, increase execution speed by compiling to machine code on the fly. Aggressive optimization, however, is scarcely possible.

[2] By host application we mean the application, in our case written in C or C++, that encapsulates and invokes the compiled MatLab code in order to do a given computation.

[3] Or, as more commonly but less precisely known, *typeless* language.

[4] Unless otherwise noted, the term *compiler* refers to the MatForce compiler implemented by the author. In order to disambiguate the MatForce compiler from the C++ compiler, the latter is always referred to as such.

the *utility library* that contains definitions for various matrix types, shapes and operations (such as product of two double-precision matrices or Fourier-transform of a vector), which makes compiled code succinct and sufficiently abstract. The utility library links against freely available BLAS and LAPACK [7] linear algebra routines written in Fortran but no other third-party libraries. As a result, the compilation process yields intuitive, type-safe, high-performance C++ source code, which the programmer can modify at will without relying on external heavy-weight libraries.

The proposed system has been successfully used in developing a UTRAN link capacity dimensioning algorithm in a mobile telecommunication network design tool.

The rest of the paper is structured as follows. Section 2 gives the motivation for using type inferring code translation as opposed to other approaches and surveys related work. A brief introduction to Prolog and DCG rules, both of which the compiler makes intensive use of, is given in Section 3. Sections 5, 6 and 7 deal with the details of the MatForce compiler, the utility library and the (external) linear algebra library, respectively. The paper ends with Section 8, which summarizes results and drafts possible future work. The reader is assumed to have a basic knowledge of the MatLab language.

2 Related work

There have been attempts at increasing MatLab performance by means of partial evaluation [9], just-in-time compilation [6], parallelization [11], direct compilation [10] and type estimation [12, 2].

Partial evaluation is a technique for program optimization by specialization, constraining code to a particular set of possible inputs, producing the so-called residual program. The MatLab partial evaluator described in [9] transforms a MatLab abstract syntax tree into a simpler but equivalent form by deducing type, shape and value range information from expressions, evaluating static subexpressions and eliminating dead code whenever possible as well as unrolling loops. This partial evaluation can substantially increase performance but produces MatLab code and hence can only serve as a possible preprocessing step in our scenario.

MaJIC [6] is a just-in-time compiler targeted at speeding up MatLab execution in an interactive environment. The compiler comprises three constituents: an analyzer that annotates code, a very fast code generator and a code repository that caches compiled code. Despite its effectiveness due to preallocating temporary arrays, eliminating unnecessary temporaries and unrolling loops, the just-in-time scheme is heavily dependent on run-time information and is therefore not suited to our needs.

Direct compilation speeds execution by translating to a compiled language such as C or C++ but by using a dynamic typing scheme. As such, it is a simple approach that may significantly speed execution by compiling control structures but it does not address type safety, nor can it exploit the speed gain from type specialization. Nevertheless, compiled code can bear close resemblance to the original, facilitating future extension.

In the case of type estimation (or type inference), variables are assigned well-defined types, allowing code to be translated into a strongly-typed language. As a result, type inference combined with code translation meets our demands both in terms of speed and type safety. Not only does type inference enable aggressive optimization of generated C++ code (by allowing cross-optimization between hosting application and MatLab algorithm) but also decreases the heterogeneity of a complex system by making it possible to use a single programming language.

In [12] a type estimator written for the Octave [8] language is described, which is very similar to MatLab in syntax and semantics. The type estimation scheme is based on flow graphs to guess the intrinsic type, size and value range of Octave matrices in an iterative process. This information is stored as a triplet and attached to each variable in an Octave program. Once the Octave program is annotated, a corresponding C++ program can be generated. In [12], both the type estimator and the generated code are dependent on the Octave runtime, an equivalent of MCR. The approach presented here, apart for being independent of the Octave (as well as the MatLab) run-time, is different in the sense that it traverses the abstract syntax tree that represents the program only once; it does not infer matrix size or value range, which are often difficult to determine for higher-level functions; it leaves more of operator disambiguation to the C++ compiler; and it puts emphasis on generating efficient yet human-readable C++ source code, such as by extensive use of iterators and pass-by-reference, which are out of scope of [12].

[2] describes a code synthesis tool that is also based on type estimation. The tool transforms MatLab code into equivalent C code (possibly merging with existing code), while taking care of memory allocation, array indexing as well as function and operator polymorphism, and the results are presented in an interactive environment. While the goals of this commercial product are similar to those presented here, it produces C rather than C++ code, the former of which lacks language support for features such as class inheritance and polymorphism, which enable resultant code to have a higher level of abstraction.

3 Background

The MatForce compiler is written entirely in Prolog. Prolog [14] is a general-purpose declarative programming language often associated with artificial intelligence and computational linguistics. It is especially suited to compiler construction because it allows easy specification of BNF grammars and formulation of relationships are rules. Declarative features such as single-assignment ensure minimal interdependence and simple fabrication of unit tests. Built-in auxiliary services, such as *indexing* (automatic

hashing on arguments) produces fast code without much intervention. Indeed, the source code of the entire MatForce compiler consists of less than 2500 lines of code, including MatLab and C++ operator and function declarations that take up a significant portion of code.

Prolog is weakly-typed, it has a single data type, which is *term*. Terms are either atoms, numbers, compound terms or variables. *Atoms* correspond to strings in most computer languages. *Compound terms* comprise a *functor* and a fixed number of positional arguments. The functor is a pair consisting of an atom that serves as the name of the term and an integer (called *arity*) that gives the number of arguments. For instance, *integer(2)* is a compound term that has the functor *integer/1* and the single argument, which is the number *2*. Atoms are special terms that have zero arity. Prolog *variables* are single-assigned: once they have been given a value through a process called *instantiation*, they are indistinguishable from whatever value they have been assigned. In program code, the name of variables is always capitalized. *Lists* have special importance in Prolog. A list is a recursively defined structure which is either the atom *[]*, which denotes the empty list, or a compound term with two arguments, the first representing an item and the second the continuation, or *tail* in Prolog terminology, denoted as *[Head|Tail]*.

Prolog programs consist of *rules* (also called *predicates*), which describe relations. Rules have the form: *Head :- Body*, where *Head* is a (possibly compound) term that is roughly the declarative equivalent of function signatures in imperative languages, while *Body* is a conjunction of other predicate calls. Unlike imperative programming languages, executing a Prolog rule can have any of the three possible outcomes: success, failure or error, the latter of which corresponds to exceptions. A rule succeeds if all calls in its body succeed, or it fails otherwise.

In terms of procedural semantics, a Prolog program is executed by specifying a goal (called the *query*), which consists of typically a single term. The Prolog engine attempts to unify the query term with the head of a predicate, scanning the program code top to bottom. If successful, the body is executed, or the goal fails otherwise. Execution of the body is done recursively as if each call it consists of were specified as a separate query one after the other with proper context information. A major difference compared to imperative languages is that a query can succeed in multiple ways because more than one head can match the query if they have the same signature. In each case when there are multiple matching heads, a *choice point* is generated. In case failure occurs, execution *backtracks* to the last choice point (undoing any variable instantiations since that choice point), from where execution is continued.

Let us take the simple example in Figure 1. Given the query *keyword(K)*, we get the results *break*, *continue*, etc. (*keyword/1* generates choice points and multiple results are obtained by backtracking). Similarly, the query *reserved(R)* yields all keywords and symbols (note the identical heads, and the bodies, the latter of which are single-element conjunctions); *reserved(integer(2))* fails and *reserved(break)* succeeds (arguments are passed as in other languages but notice that it is possible to pass entire terms).

```
keyword(break).
keyword(clear).
keyword(continue).
...
symbol('(').
symbol(')').
...
reserved(Item) :- keyword(Item).
reserved(Item) :- symbol(Item).
```

Figure 1. A sample from the Prolog definition of the MatLab language in MatForce.

An extremely useful feature intrinsic to Prolog is the definite clause grammar (DCG) [13] formalism. DCG rules allow fast and effective coding of tokenizers, parsers and code generators. A DCG rule has the following form, bearing close resemblance to BNF rules: *Head --> Body*, where *Head* is a regular predicate head and *Body* is made up of DCG terms. Each DCG term may either be a list of token terms (enclosed in brackets), a DCG predicate or a regular Prolog predicate call (enclosed in curly braces). The first argument of *Head* is generally an abstract syntax tree which results from parsing the specified token terms and evaluating the intermitting calls.

```
digit(C) -->
   [C], { C >= 0'0, C =< 0'9 }.
digits([H|T]) -->
   digit(H), digits(T).
digits([]) --> [].
number(Number) -->
   number_digits(Digits),
   { number_codes(Number, Digits) }.
```

Figure 2. A sample from the MatForce tokenizer.

Consider the excerpt from the MatForce tokenizer in Figure 2, which has been slightly modified for clarity. Here, *digit* extracts a single digit from a stream of characters in the range 0–9, *digits* extracts as many digits as possible, while *number* converts the series of digits into an integer. Notice how the digit list is built implicitly in the head of *digits*.

4 Overview

The MatForce system comprises of three easily identifiable parts. The *compiler* performs the MatLab to C++ translation with type inference, yielding C++ code. The *utility library* contains the implementation of matrix operations. The C++ compiler translates both the utility library and the C++ source code produced by the MatForce compiler into object files. The linker binds the object files to the high-performance *linear algebra library*, which carries out computation-intensive operations. (Figure 3)

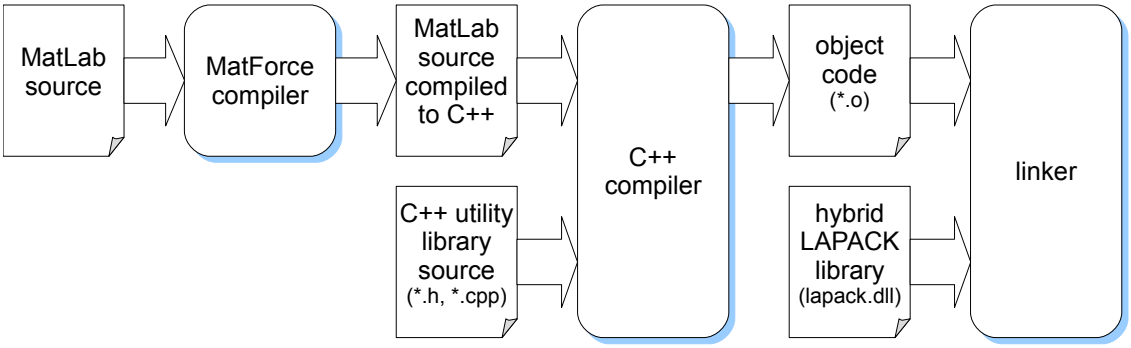

Figure 3. The MatLab to C++ compilation and link process

5 The compiler

The MatLab code *compiler*, written in SWI-Prolog[5] [5], is the major constituent of the system. Compilation is done in four stages: tokenization, parsing, type inference and code generation.

Tokenization At this stage, m-files are read and converted into a stream of tokens by means of predicates similar to those shown in Section 3. The term *token* may refer to an integer, a real number, a keyword, a string, a literal, a symbol (which corresponds to a MatLab operator), a comment or a newline. Keywords and symbols, which are built-in MatLab constructs, are represented by Prolog atoms, while other tokens are represented as single-argument terms. Line folding is also performed at this stage. Tokenization uses some but little context information.[6]

Parsing During the parsing process, tokens are interpreted in their context and an abstract syntax tree is built. Each m-file contains one or more *functions*, each of which comprises multiple *statements*. A statement may be a control statement (*return*, *break*, *global* etc.), an assignment, an *if*, a *while* or a *for* statement, or a function call. The exact semantics of each statement is defined by means of the DCG formalism. For instance, a *while* statement is defined as in Figure 4.

The DCG rule in Figure 4 states that a *while* statement begins with the *while* keyword (Line 2) and is followed by an expression (Line 3) of conditional (boolean) type (Line 4). The *while* statement wraps a group of other MatLab statements (Line 6) and the group is terminated by the *end* keyword (Line 7). Line 9 defines the abstract

```
(1) while(Context, NewContext, Loop) -->
(2)   [while],
(3)   expression(Context, Context1,
                 Conditional, Types),
(4)   { conditional_type(Types) },
(5)   closing_comma, [eol],
(6)   statements(Context1, InnerContext, Statements),
(7)   [end], !,
(8)   { chain_context(Context1, InnerContext,
                      NewContext) },
(9)   { Loop = while(Conditional, Statements) }.
```

Figure 4. The DCG definition of a *while* statement

syntax tree that belongs to the *while* statement: a node that contains the conditional and encapsulates nested statements as a list of branches.

Type inference As MatLab is a dynamically-typed language, variables are not explicitly associated with types. In contrast, their type is determined run-time using the implicit type of the initializer or assignment expression, and that type is changed as necessary whenever another expression demands so. In our case, a simplifying (but scarcely restrictive) "same type" assumption is made, that is, a variable cannot have incompatible types in the same MatLab function. For instance, if a variable *i* has been concluded to have the type *integer*, it cannot be part of an expression where it should be interpreted as a *string*, which is not more general than *integer*.

Having made the "same type" assumption, the compiler uses a domain narrowing technique, common to constraint logic programming (CLP), to infer the type of a variable. Once a variable has been introduced, it is associated with a type domain. Initially, the domain contains all possible MatLab types. Types are classified into

- intrinsic types; *boolean*, *integer*, *real* and *complex*
- primitive types; all intrinsic types and *string*
- matrices and vectors, which are made up of elements of the same intrinsic type

[5] SWI-Prolog is an ISO-compliant Prolog system. As the special features of SWI-Prolog are exploited only to a very limited extent, the source code is expected to be easily portable.

[6] MatLab uses ' (apostrophe) both as a string terminator and as the conjugate transpose operator, which have to be distinguished. An apostrophe is interpreted as a postfix operator after a literal, but as a string starter character in any other context (because it starts an operand).

Narrowing occurs in any of the following two situations:

1. *Direct assignment*. The variable is assigned an expression. For instance, the assignment *a = 1 + 2* allows the compiler to infer that *a* is an integer, a real or a complex number; *a = b + 4* leads the compiler to conclude *a* is a real (or complex) matrix if *b* is a real matrix.

2. *Implicit from context*. The variable is used in an expression in a context that allows deductions to be made as to the type of the variable. For instance, the assignment *a(6,b) = 11* implies *a* is a matrix because only matrices can be accessed with two indices. Similarly, the compiler can conclude that *b* is an integer or an integer vector: other types are not allowed as indexers.

In each case, the domain of the variable is the intersection of its current domain and the domain deduced from the expression. If the domain of a variable becomes the empty domain, the compilation halts with an error. Ideally (and in most real-world situations where code is well-formed) the domain reduces to a single type by the end of the compilation unit. However, notice that there is a hierarchy of types: e.g. the type *integer* is a special kind of 1-by-1 matrix. As the type inference algorithm excludes types only when they are inappropriate, choosing the simplest type will not harm code semantics. In particular, counters or indexers will not be treated as general matrices.

As previously seen, inferring the type of an initializer or an assignment requires inspection of the assigned expression. This is recursive: the type of the expression itself is determined chiefly by the type of its subexpressions. The recursion is terminated by constants (which have a definite type) and variables (whose type may also depend on other external expressions). MatForce uses Prolog backtracking to infer the possible types of an expression. First, an expression type tree it built, which is almost identical to the original expression abstract syntax tree except that constants are replaced by their type and variables by (uninstantiated) Prolog variables. Second, each variable is instantiated with a type from its domain. Third, the resultant type tree, which now has no variables (i.e. it is *ground*), is simplified into a single leaf using operator and function type transformation rules (e.g. real + complex = complex). The remaining leaf constitutes the type of the entire expression. Notice that there are multiple ways the second step can be performed (i.e. it leaves Prolog choice points) and that the simplification may fail. These, however, seamlessly fit into the Prolog execution model.

Let us consider an example in which two variables *a* and *b* have the type domain *[boolean,integer]* and *[integer,float]*, respectively. In order to infer the type of the expression *a + b + 1*, MatForce first builds the following type tree: *A + B + integer*. Second, the variables *A* and *B* are instantiated with types from their domain, e.g. *boolean* and *integer*. Third, a simplification is attempted using the ground term *boolean + integer + integer*. However, the operator + cannot take arguments of type *boolean* and *integer*. Consequently, execution backtracks, and a second instantiation is attempted. The substitution $A \leftarrow integer$, $B \leftarrow integer$ is successful, and the type *integer* is inferred. Similarly, $A \leftarrow integer$, $B \leftarrow float$ is a valid instantiation pattern, yielding the expression type *float*. As a result, the expression will have the type domain *[integer, float]*. In parallel, the domain of *A* will have been narrowed to *[integer]*, as its being *boolean* would not produce a valid expression in terms of type.

It is important to emphasize that a particular type inferred for an entire expression depends on the variable type substitution pattern in that expression. When all inferred types are aggregated into a single type domain, information in the actual substitution pattern is lost. In order to overcome this phenomenon, possible outcomes for the type of an expression are also recorded as Prolog rules in terms of the type of relevant variables. When the final type for the expression is required at the code generation stage, each candidate type in the domain is tested against these rules. If the domain of a relevant variable has been found to be actually narrower than used to infer the type of the expression, the corresponding types in the type domain of the expression will be dropped.

Code generation In the last compilation stage, the type-annotated abstract syntax tree is converted into a series of C++ expressions. The compiler takes operator precedence into account in order to use the least number of parentheses for improved legibility. Some of the simplifications and optimizations at this stage include using increment and decrement operators, simultaneous arithmetic and assignment (e.g. add-and-assign), and iterators in loops.

Although the current implementation uses C++ as the target language, the code generator is a relatively minor part of MatForce. With small or medium effort, the generator can be fitted to other programming languages, such as C# or Java.[7] The entire code generation process is driven by DCG rules, which can, in most part, be easily fitted to the needs of a different language.

6 The utility library

The *utility library*, written in C++, exposes a strongly-typed class hierarchy of different types of vectors, matrices and 3-dimensional arrays. Matrix types are differentiated based on the intrinsic type they store, which can be unsigned integer, signed integer, double-precision real or complex numbers.[8] Generic functionality, common to more types and shapes, is implemented in corresponding storage and matrix classes. Operators are overloaded to provide a convenient way to express arithmetic operations.

[7] However, generating code in another language assumes a corresponding utility library in that language.

[8] The utility library has limited support for multiple-precision real matrices if compiled with GMP [3] support.

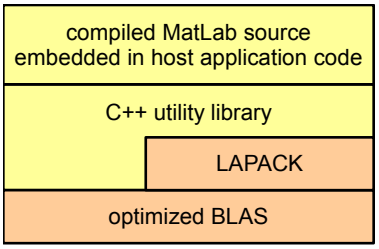

Figure 5. The structure of an application compiled with MatForce

While simple operations (such as computing the element-wise minimum of two matrices) are implemented directly in C++, more complex operations (such as matrix multiplication or inverse) are performed by the underlying linear algebra library.

The major design goals of the utility library were support for extension at the C++ level and efficiency of compiled C++ code, which is the rationale behind intensive use of operator overloading. Matrix arithmetic can be read in familiar notation and additional required code can hence be easily interleaved.

7 The linear algebra library

The *linear algebra library*, written in Fortran, serves as the fundamental layer for high-performance computation. It contains efficient implementation of such operations as matrix multiplication, matrix decomposition or eigenvalue computation. The linear algebra library itself comprises two layers: the lower BLAS (Basic Linear Algebra Subroutines) layer and the higher LAPACK (Linear Algebra PACKage) [7] layer. The most critical part of the library in terms of efficiency is the BLAS layer, which implements basic vector, matrix-vector and matrix-matrix operations, all of which are used both by the LAPACK layer and the C++ utility library (Figure 5). ATLAS [1, 15], which is an automatically tuned BLAS (and partial LAPACK) implementation, is used to exploit the specific features of the target computer's hardware. ATLAS and (the unoptimized part of) LAPACK have been combined in a self-contained dynamically-linked library, which can also be used independently of MatForce.

8 Conclusions and future work

In this paper, a type-inferring compiler that converts MatLab scripts to C++ source code as well as an accompanying utility library has been presented. Albeit it uses optimized open-source BLAS and LAPACK routines written in Fortran for the sake of efficiency, the compiled code does not have any external dependencies on third party proprietary libraries. The generated C++ code is properly annotated with types, using primitive types whenever possible. The code is fully legible from a human perspective using matrix and vector types as well as related operators, which allows further additions or modifications to the translated code.

Future work includes more effective (potentially user-guided) type narrowing, a user interface to control the compilation process, support for a greater proportion of MatLab functions in the supplemental C++ utility library, and a study of comparative performance w.r.t. other approaches.

References

[1] Automatically Tuned Linear Algebra Software, *http://math-atlas.sourceforge.net/*

[2] Catalytic MCS Family: MATLAB to C Synthesis, *http://www.catalyticinc.com/*

[3] The GNU Multiple-Precision Library, *http://gmplib.org/*

[4] SourceForge.net project site of MatForce, *http://matforce.sourceforge.net/*

[5] SWI-Prolog, *http://www.swi-prolog.org/*

[6] George Almasi, David A. Padua, MaJIC: A Matlab Just-In-Time Compiler, in S. P. Midkiff et al. (eds.), *LCPC2000*, LNCS 2017, 68-81, 2001

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide (Third Edition), Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999, ISBN 0-89871-447-8 (paperback)

[8] John W. Eaton, Octave: Interactive language for numerical computations, 2007, *http://www.gnu.org/software/octave/doc/interpreter/*

[9] Daniel Elphick, Michael Leuschel, Simon Cox, Partial Evaluation of MATLAB, in F. Pfenning and Y. Smaragdakis (eds.), GPCE 2003, LNCS 2830, pp344–363, 2003

[10] Y. Keren, MATCOM: A MATLAB to C++ Translator and Support Libraries, Israel Institute of Technology, 1995

[11] S. Pawletta, T. Pawletta, W. Drewelow, P. Duenow, M. Suesse, A MATLAB toolbox for distributed and parallel processing, in C. Moler, S. Little, ed., Proc. of the Matlab Conference, Cambridge, MA, MathWorks Inc., October 1995

[12] Jens Rcknagel, An Octave Type Estimator – Essential Part of an Octave to C++ Compiler, Thesis, Technical University Ilmenau, Faculty of Computer Science and Automation, Department of System and Control Theory, Ilmenau, April 12, 2005, *http://www.stud.tu-ilmenau.de/~ rueckn/*

[13] C. M. Sperberg-McQueen, A brief introduction to definite clause grammars and definite clause translation grammars, A working paper prepared for the W3C XML Schema Working Group, 18 July 2004, *http://www.w3.org/People/cmsmcq/2004/lgintro.html*

[14] Leon Sterling, Ehud Shapiro, The Art of Prolog: Advanced Programming Techniques, MIT Press, Cambridge, Massachusetts, 1994

[15] R. Clint Whaley, Antoine Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience, volume 35, number 2, pp101–121 February 2005, *http://www.cs.utsa.edu/~ whaley/papers/spercw04.ps*