# FALCON: An Environment for the Development of Numerical Programs Using MATLAB

L. DeRose* B. Marsolf† K. Gallivan‡ E. Gallopoulos§ D. Padua*

**Abstract**

This paper describes FALCON, an environment for the development of numerical programs using MATLAB. FALCON supports an algebraic language to provide the developer the freedom of algorithm representation and reduced development time. This is combined with a compiler and transformation system to convert the algebraic language into source code which can call other programs and libraries developed with the system or independently and which can be embedded in an application code. During the compilation and transformation process, the developer is able to interactively apply optimizations and transformations to the code, including both traditional compiler techniques and other transformations which utilize algebraic information about the operations performed and target libraries in which they are implemented. The environment includes modules that allow MATLAB programs to be analyzed and then optimized, both automatically and semi-automatically. Fortran 90, C++, or restructured MATLAB code implementing the transformed algorithm can be produced by the code generators included in the environment.

## 1 Introduction

Scientific applications rely upon numerical algorithms to effectively utilize high-performance computers; however, the steps necessary to provide an application developer with efficient algorithms can be very complex, therefore problem solving environments (PSEs) that help the user in this endeavor are desirable. A typical scenario has the user who needs to accomplish some problem-solving task (e.g. consisting of input, simulation, and output steps) and in response designs a solution method and implements it on the target computational platform(s). In many instances the above process is repeated until an effective program or routine is obtained. It is thus important that the environment offers facilities for *rapid prototyping*. On the other hand, users would like *high performance* on the target computational platform(s), especially since often the advantages of one method compared to another become visible only for large data sets.

*University of Illinois at Urbana-Champaign. This work was supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

†DEMACO, Inc., Champaign, Illinois. This work was supported in part by the National Science Foundation under Grant No. US NSF CCR-9120105 and by ARPA under a subcontract from the University of Minnesota of Grant No. ARPA/NIST 60NANB2D1272.

‡University of Illinois at Urbana-Champaign. This work was supported in part by the National Science Foundation under Grant No. US NSF CCR-9120105 and by ARPA under a subcontract from the University of Minnesota of Grant No. ARPA/NIST 60NANB2D1272.

§Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece. This work was supported in part by the National Science Foundation under Grant No. US NSF CCR-9120105 and by ESPRIT Copernicus RTD Project STABLE No. 60237.

Approaches that help the computational scientist in these tasks include tools for automatic language translation such as Parafrase-2 and Polaris, annotation systems for conventional languages such as PC++, symbolic, algebraic and numerical computing systems, like Mathematica, MAPLE and MATLAB, high-level language environments for partial differential equations, and such as Ellpack and Diffpack, and application specific PSEs such as CTDEL. The convenient features of high-level rapid prototyping tools, however, frequently come at the cost of computational performance. Commenting on this matter, it was suggested in [GHR92][1] that it would be desirable to build a system to automatically create "standard" high-level language output from programs written in the language of the PSE and subsequently take advantage of mature compiler technology to obtain high performance; and to exploit the environment's high-level knowledge of the problem to enhance the compilation process. We have thus designed and implemented FALCON (FAst Array Language COmpilatioN), an environment for the development, support, and use of high-performance numerical programs and libraries that combines the software techniques from compilers with the algebraic techniques used by algorithm developers [DGG+94]. FALCON supports an algebraic language to provide the developer the freedom of algorithm representation and quicker development time. This is combined with a compiler and transformation system to convert the algebraic language into source code that can call other programs and libraries developed with the system or independently and which can be embedded in an application code. During the compilation and transformation process, the developer is able to interactively apply optimizations and transformations to the code, including both traditional compiler techniques and other transformations which utilize algebraic information about the operations performed and target libraries in which they are implemented.

To support the algebraic form, the interactive array language of MATLAB[2] is used [Mat92a]. The advantages of this language is that first, it is not necessary to specify the dimension, shape, or type of elements of arrays allowing algorithms to be coded more quickly. Second, this language has an extensive set of functions and higher level operators that facilitate the development of scientific programs by easing the problem of interfacing to existing libraries and whose algebraic properties are understood and can be used for optimizations. Both of these capabilities make MATLAB an excellent choice for algorithm development and it is currently used extensively by computational scientists on a wide range of platforms. Furthermore, by FALCON supporting the data file formats of MATLAB, it is possible to exchange data between the two systems, taking advantage of the capabilities of both systems, see Figure 1 for a example use of MATLAB.

FALCON provides the capability for transforming the interactive array language into a more traditional language, with the ability to apply transformations to the code. Such transformations may range from traditional code optimizations, such as common subexpression elimination and constant propagation, to the loop based transformations, such as loop interchanging and loop unrolling. Just as important, however, is the compiler's ability to utilize the extra algebraic information which is available to the environment during the transformations. In addition, multiple output languages can be targeted, with the current choices being Fortran 90, C++, and MATLAB.

## 2 Related Work

Several approaches are presently used to improve the performance and development process of numerical applications, ranging from restructuring compilers to application-specific environments. These approaches, however, tend to only focus on one aspect of the development and do not address the issues involved over the life-cycle of the application.

[1] An abbreviated form of that report appeared in the Spring'92 issue of IEEE CS&E

[2] FALCON uses language elements present in version 4.2c of MATLAB.

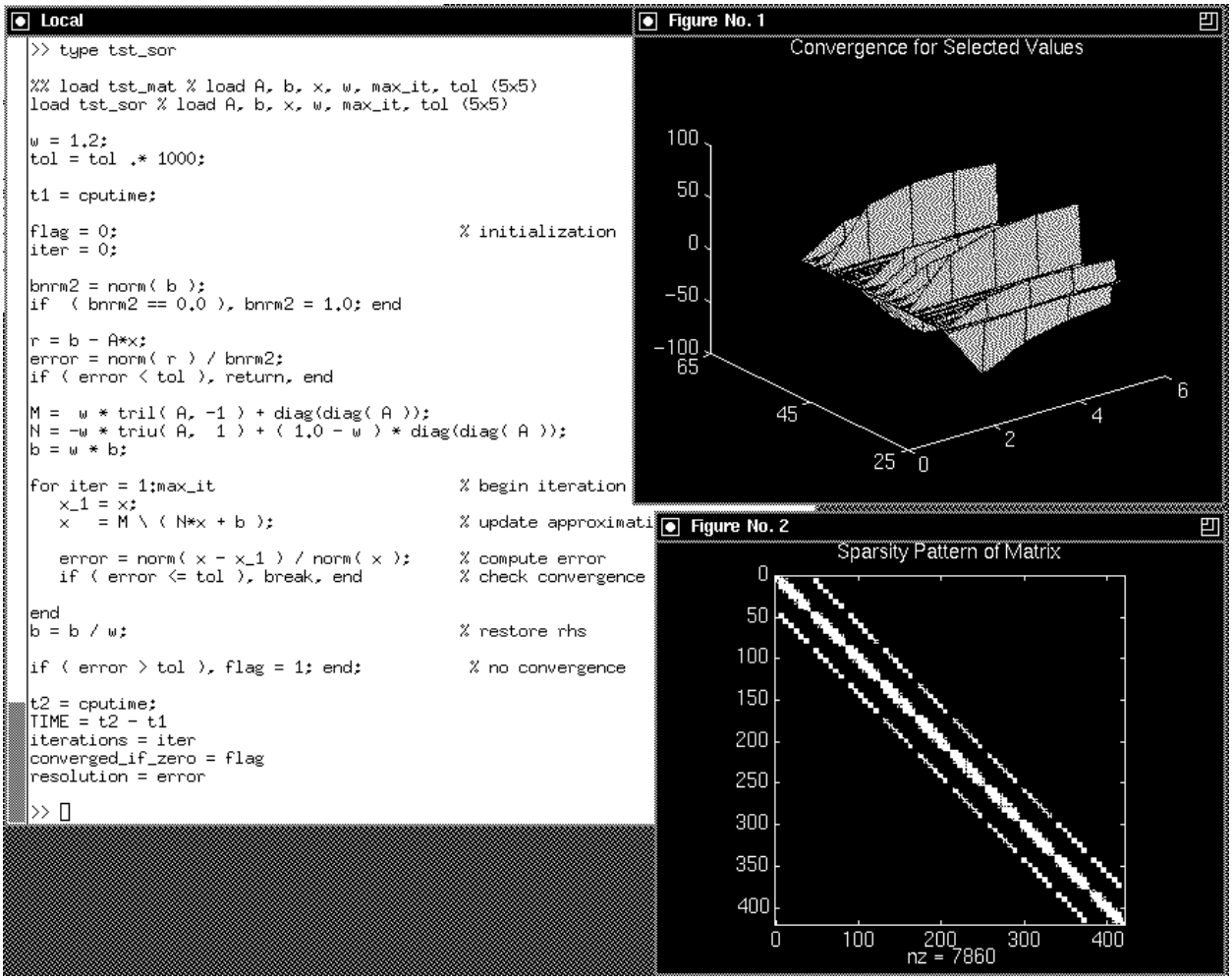


Figure 1: Example use of MATLAB environment.

Restructuring compilers are able to optimize the numerical operations that are present in the code for both scalar and array operations; see [Wol96] and references therein. Most of these optimizations, however, are based on recognizing a single expression or recognizing a scalar expression contained within nested loops, where typically the loop corresponds to a matrix expression or a reduction operation. As a result, more aggressive algorithmic optimizations are usually impossible due to the lack of understanding of the essential algorithmic intent. This is, of course, due to the inevitable loss of key information inherent when expressing the algorithm in a particular programming language. Carr and Kennedy have worked on compiler optimizations to automatically block numerical algorithms for dense linear algebra on machines with memory hierarchies [CK92]. Even for this more specific restructuring task, they have encountered the fundamental difficulty of many high performance algorithms: the best block algorithms cannot always be derived solely based on the operations and computational primitives which are present in the code. Gallivan et al. [GPS90] have pointed out that it is often possible to move easily from standard algorithms with limited parallelism to highly parallel algorithms by using "higher level transformations" that exploit algebraic knowledge of the operations. It is this merger of high level transformations, algebraic knowledge, and more traditional compiler strategies applied to an algebraically expressive language that is to be exploited by the restructuring techniques within the FALCON environment.

The approach in [BBC+93] is to distribute only the algebraic algorithms (called templates), instead of code, and allow the implementor to make the required optimizations. While this approach keeps the algorithms portable, it requires the developer to perform the transformation from algorithm to code by hand and, once transformed, the algorithm may no longer be portable. This approach forces the user to deal with all of the implementation decisions while allowing the numerical algorithm designer to ignore most architectural details.

We also note the various efforts on object oriented programming techniques and systems for scientific computing, such as the Diffpack project [DT97]. The approach in [GS93] has concentrated on standardizing the interface to the library's algorithms and data structures. This allows both algorithms and data structures to be interchanged with almost no changes to the calling program. While this approach can help the developer of application programs try different algorithms, it constrains the developer to a limited set of algorithms and data structures and it is the responsibility of the library designer to incorporate new algorithms or data structures into the system. Furthermore, the new algorithms and data structures must be written to fit predefined standard data access patterns.

One difficulty with using a programming environment like MATLAB is that the flexibility within the environment is provided by using an interpreter to execute the commands. The overhead incurred by the interpreter for loop control structures, the element-wise access of arrays, and dynamic memory allocation can have a significant impact on the performance of the algorithm. On the other hand, array-level operations are typically implemented using high-performance subroutines allowing codes consisting of mainly array operations to run efficiently.

In order to overcome the performance cost of the interpreter, one option is to translate the algorithm into a language that can be compiled. This is the approach that was taken by the FALCON environment and is also being considered by others. The developers of MATLAB, The MathWorks, Inc., have developed a compiler, MCC [Mat95], that translates MATLAB algorithms into C programs. Another project, MATCOM [Yar96], translates MATLAB algorithms into C++ programs. The goal of both of these projects is the translation of the MATLAB code into an equivalent C or C++ program. With the FALCON environment, however, the goal is to be able to produce multiple equivalent programs, based upon transformations and optimizations selected by the algorithm developer. Two other MATLAB projects of note specifically geared to parallel processing are CONLAB [JKR92] and MultiMATLAB at Cornell, whose purpose is to to run MATLAB in parallel without translation to a compilable language; see the recent MATLAB conferences for more details.

# 3 FALCON Overview

The FALCON environment consists of three main components, namely the Program Analysis System (PAS), the Interactive Restructuring System (IRS), and the Code Generation System (CGS), as shown in Figure 2. This figure presents an overview of the development environment, with the FALCON components being contained within the shaded box, and illustrates the flow of information between the main system components.

While the type-less nature of the MATLAB language facilitates the prototyping of algorithms, it greatly hinders the translation into traditional programming languages that require variable declarations. Inference techniques are used to analyze the algorithm and attempt to statically resolve the type, rank, and size of the variables. When the techniques fail, code is inserted to complete the analysis at run time. However, the IRS can be used to interactively supplement the analysis and remove this code. During this program analysis phase, the MATLAB program is

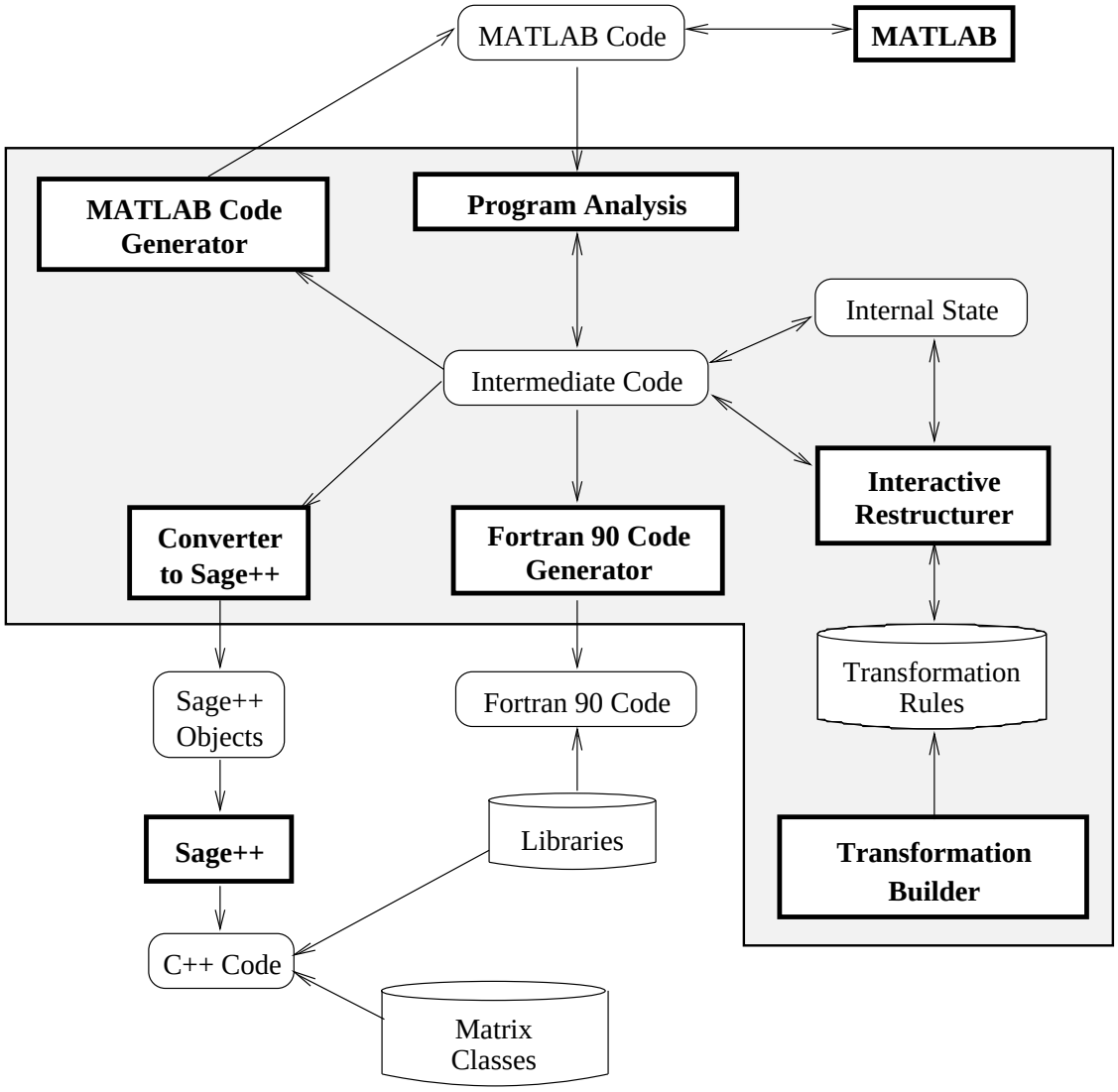


Figure 2: Overview of the FALCON environment.

translated into an intermediate form.

After the program analysis phase, the IRS can be used to apply both compiler and algebraic techniques to the algorithm. The transformations, based on a pattern-matching system, can be performed either automatically or interactively.

After transformation, the intermediate form is translated into one of three output languages: Fortran 90, C++, and MATLAB. Fortran 90 was a natural choice as Fortran has traditionally been used for numerical programming. C++ was chosen as an output language both because of its growing popularity, especially in the field of numerical analysis, and because of the flexibility that arises from its use of classes for defining the matrix operations. The third language choice, MATLAB, was chosen both because there are optimizations that can be made to improve the MATLAB code and because it allows the transformed code to be saved for later reuse in the environment. In the next few sections we outline the key features of FALCON and refer to [DeR96, Mar97] for detailed discussions of each.

# 4 Program Analysis Phase

The Program Analysis System [DeR96] reads in the MATLAB program and converts it into an abstract syntax tree (AST), which is the intermediate form for the environment. Before the AST is generated, all function calls to M-files, subroutines written in MATLAB's array language, are in-lined in the program. In-lining makes it easier to perform the program analysis on a function for the correct combination of input variables, as subsequent calls to the same function may have input variables with different types or ranks.

During the transformation to the AST, the system analyzes the program to determine *variable properties*, such as type (i.e., COMPLEX, REAL, INTEGER, LOGICAL, or STRING), rank (e.g vector, square matrix, scalar, etc.), and shape (i.e. the size of each dimension). This work built upon type inference techniques developed for SETL [Sch75] and APL [Bud88]. The inference techniques were also extended with techniques originally developed to analyze array accesses within Fortran loops [TP93]. The analysis utilizes a *high-level summary* of the behavior of the built-in MATLAB functions for the various combinations of input variable properties. This high-level information increases the accuracy of the analysis over the approach used in conventional high-level language compilers, where only the information from elementary operations is used.

## 4.1 Use-Definition Coverage

The MATLAB program is internally represented in Static Single Assignment (SSA) form [CFR+91], whereby a new version of the variable is used for each assignment statement. This is a convenient representation for implementing many of the analysis algorithms that are used within the environment. The standard SSA representation was enhanced to handle the case of arrays, whose definition coverage is clearly more complex than scalars.

## 4.2 Automatic Detection of Variable Properties

To generate variable declarations and to support data structure selection, the system infers variable properties using a forward/backward scheme. Although described separately here, the inference of the shape of variables works in coordination with the coverage analysis discussed in the previous subsection.

The inference is performed on the AST in a sequence of static and dynamic phases; static inference determines which variable properties are known for the program given the semantics of the statements and, possibly, using information about the input variables. Upon completion of the static inference phase, the dynamic phase inserts, into the AST, the code necessary to resolve any unknown variable properties at run time.

Type inference uses a *type algebra* similar to the one for SETL [Sch75]. This algebra operates on the type of the MATLAB objects and is implemented using tables for all operations and built-in functions. Each node in the AST contains attribute fields to store the inference information. These fields are filled during the static inference phase and propagated through the AST whenever a new attribute is synthesized. If these attributes are inferred to have different (or unknown) types, then the node is marked to be resolved during the dynamic phase, or a type that subsumes all possible types is assigned to the variable. Array shapes are estimated using induction variable and range propagation algorithms. Again, in the case of rank and shape, code to apply dynamic analysis is generated whenever the static analysis fails.

### 4.3 Run-Time Determination of Variable Properties

In some cases not only is the static information insufficient, but the developer is also unable (or unwilling) to provide enough information to determine variable properties. To handle such cases, FALCON generates code to determine missing properties at run time in order to allocate the necessary space and select the appropriate code sequence. Tags, stored in *shadow variables* for type and shape, are associated with each variable of unknown type or shape and are tested by conditional statements in order to select the appropriate operations and allocate the necessary space.

Run-time tests of the shadow variables are used to perform the associated numerical operations correctly and efficiently. For example, when two variables, `A` and `B`, are multiplied, but the rank of neither is known, the operation can fall anywhere in the BLAS cube shown in Figure 3 where the axes represent BLAS1 operations that combine vectors and scalars, the faces represent BLAS2 operations that combine a matrix with vectors and scalars, and the interior that represent BLAS3 operations that combine matrices. If, in the same example, the type had been unknown instead,

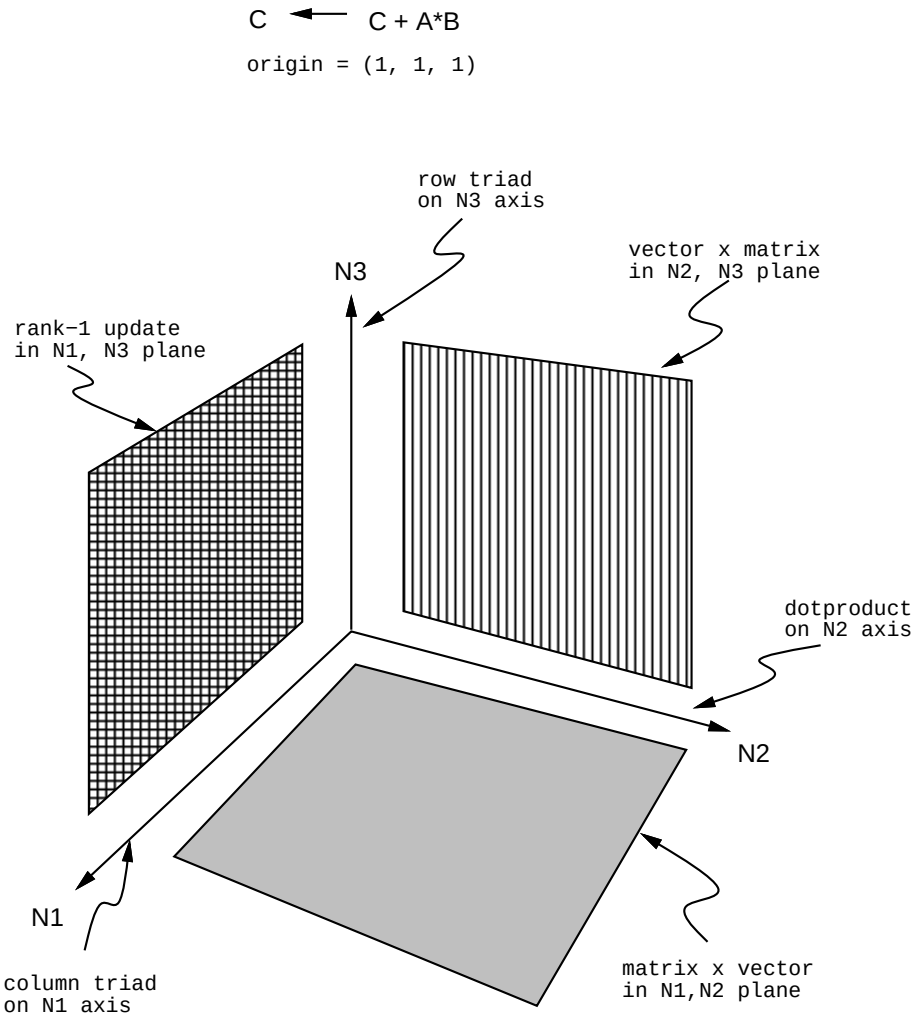


Figure 3: The BLAS operation cube.

then there would be four possible operations:

- REAL `A` times REAL `B`,

- COMPLEX `A` times REAL `B`,
- REAL `A` times COMPLEX `B`, and
- COMPLEX `A` times COMPLEX `B`.

Clearly, expanding the code to perform the run-time tests cannot be done indiscriminately. For example, if `C` is a scalar of unknown type, it may be faster to assume throughout the program that it is a complex variable and generate code accordingly. However, if `C` is a large array, or if it is a scalar that is operated with large arrays, the overhead of the `if` statement will be minimal compared to the extra cost of the assignment, if `C` is often a real.

Also, in the case of right-hand sides with many operands, or in the case of loops containing many statements, the number of rank and type combinations would grow exponentially. This problem is reduced by transforming the long expressions into triplets, or by distributing the loop; in some cases, however, the best strategy may be to assume that the variable is `complex`.

An additional issue to be considered when inserting run-time tests is whether all the tests are actually needed. After there are enough tests to guarantee that the correct numerical operation will be performed, the additional tests are typically used to select the algorithm that is optimized for a particular combination of operand ranks. Consider the previous example where `A` and `B` of unknown ranks are multiplied. Once it is known that neither `A` nor `B` is a scalar, then a matrix-matrix multiplication algorithm could be used for the other five possible combinations. The use of less tests could significantly reduce the code size and make it easier to maintain and modify the generated code, however, with a possible decrease in performance.

# 5 Restructuring Phase

The FALCON Interactive Restructuring System (IRS) [Mar97] was designed to provide the algorithm developer with interactive techniques for use during the development of numerical algorithms by combining the software techniques from restructuring compilers with the transformations utilized in numerical code development.

The IRS is built around pattern-based transformations so that the system can be tailored or enhanced to fit the individual needs of the developer. To provide this support, there are four main components within the system. First, there is the user interface that controls the application of transformations and can be used to examine properties of variables. Second, there is the transformation module for applying the transformation patterns and other restructuring algorithms and for propagating variable information. The database of transformation patterns is the third component. And fourth, the program for encoding the patterns into the internal format.

Another need for flexibility within the restructuring system comes from the desire to perform transformations at multiple levels. At one level there are the transformations that reorder the statements within the internal format, but only utilize operations that can be represented in MATLAB. However, at a second level, there are transformations that insert into the code being developed operations found only in the target language.

All interactive operations are controlled from the main user interface, shown in Figure 4. This main window consists of three parts: a row of control buttons across the top, a text screen on the left side which shows the current state of the code being translated, and a text screen on the right side which shows the available transformations. In the subsequent sections, the two main uses of the IRS, the refinement of the data analysis and code transformations, will be described.

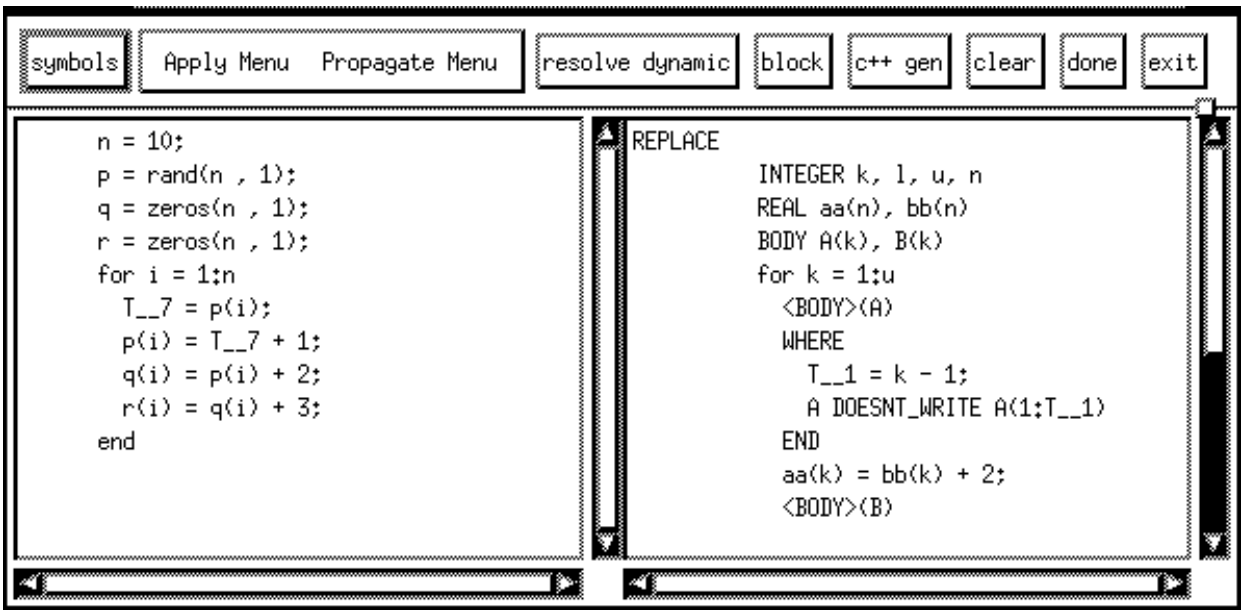


Figure 4: Main user interface to Interactive Restructuring System.

### 5.1 Refinement of Analysis

The first button on the main user interface, "symbols", is used for accessing the symbol information screen, shown in Figure 5. This window shows all the symbols currently defined in the symbol table for the program and lists three variable properties for each symbol: the intrinsic data type, the variable rank, and whether the variable has a user-defined type.

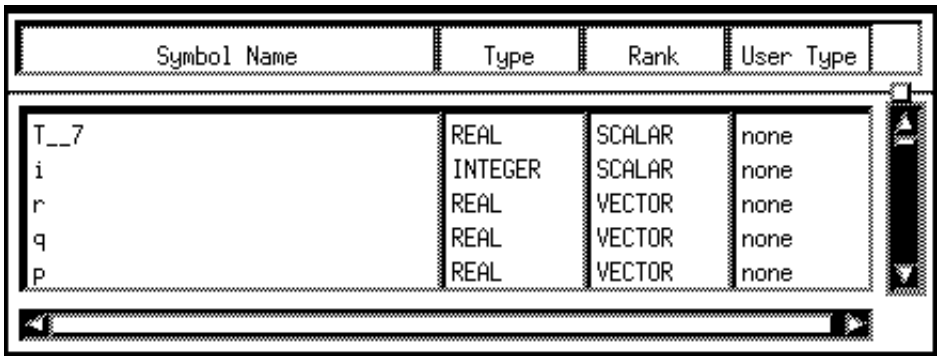


Figure 5: Window for displaying symbol information.

When a variable in the symbol information screen is selected, the symbol update window is displayed, from which the variable properties can be manipulated. The symbol update window, shown in Figure 6, provides the variable name and three columns of selections; each column corresponding to the possible values for one of the three variable properties: intrinsic data type, variable rank, and user-defined type. When the screen appears, the current properties of the variable are selected. The user can then modify the properties by the selection other values. Once the user is finished, the new properties of the variable can be updated by selecting the "apply" button, or discarded by selecting the "cancel" button.

After the properties of the symbol, or symbols, have been updated, the changes to the symbols

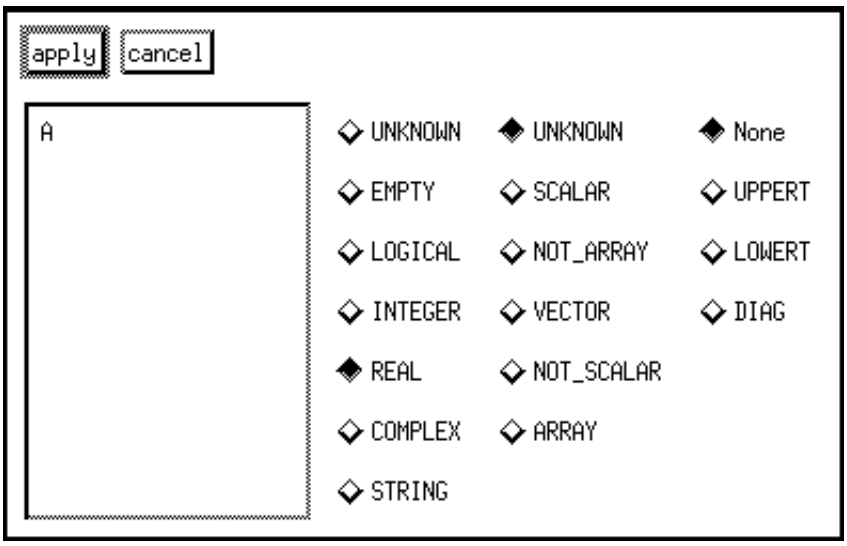


Figure 6: Window for updating symbol information.

can be propagated through the program, updating the program state and the properties of other variables, through the "Propagate Menu" on the main window.

## 5.2 Interactive Code Transformations

Transformations to the source program can be applied automatically or individually to all or part of the source program. The application of the transformations is controlled through the "Apply Menu" of the main window (see Figure 4). To apply all transformations to all of the source program, the "apply all patterns" item in the "Apply Menu" is selected. To apply one transformation to a particular section of code, the main window of the interface is used. In the left screen, the section of the source program where the transformations is to be applied is selected. In the right screen, an individual transformation is selected. To apply the transformation, the "apply selection" item of the "Apply menu" is selected.

### 5.2.1 Transformation Patterns

Unlike traditional compilers, in which transformations are essentially hard coded, FALCON uses a language to define transformations that are applied by means of a pattern-matching system and are stored in a special database. In order for the transformation language to be flexible, it supports all the source as well as the target languages. Selection and replacement is done based on code patterns whose basic form is shown in Figure 7. Three of the keywords that have been added to the transformation language are seen. The block of statements between the `REPLACE` and `WITH` keywords are used to represent the source code fragment to be selected and is referred to as the REPLACE block. The block of statements between the `WITH` and `END` keywords are used to represent the modification or replacement part of the transformation and is referred to as the WITH block. In both cases, the block of statements consists of two parts, variable declarations and transformation statements. We note the similarities of this approach for describing transformations with the approaches used by the Forbol system [Wea94] and the TXL system [CC93]. In both of these systems, however, the the transformations are meant to be written by the compiler writer, where as in FALCON, the user is able to write new transformations and insert them into the system.

```
REPLACE
      variable declarations
      transformation statements
WITH
      variable declarations
      transformation statements
END
```

Figure 7: Basic form of the transformation pattern.

The transformations supported by the system can be divided into three categories: compiler transformation, algebraic transformations, and code generation transformations. The first two types are transformations are used primarily to translate the MATLAB operations into other MATLAB operations, while the third type, code generation transformations, are used to translate the MATLAB operations into specific operations in the output language.

### 5.2.2 Variable Declarations

Whereas MATLAB has no variable declarations, the variable's type and rank are determined implicitly or at run time, the transformation language requires all variables (except for automatically-generated temporaries) to be explicitly declared. This is necessary because certain transformations only hold for specific variable type and rank combinations.

A variable declaration consists of the following four main fields:

*type* *name* ( *index* ) { *properties* },

with the first two fields required and the last two optional. Multiple variables may be declared on the same line by placing a comma after the first declaration and then appending the next declaration, excluding the type field. All variables on the same line are assumed to be of the same type.

The three variable types supported within the transformation language are `INTEGER`, `REAL`, and `COMPLEX`. These type names correspond to the types recognized by the Program Analysis System. The variable name may be any variable name supported by MATLAB, with the exception of the keywords reserved for the transformation language. The rank of the variable is determined by placing an index set, within parentheses, after the variable name. If the index set is missing, then the variable is assumed to be a scalar. If the index set contains a single value, then the variable is assumed to be a vector. And, if the index set contains two values, separated by a comma, the variable is assumed to be a matrix.

The final field in a declaration, *properties*, is used to set variable properties that are difficult to represent using the other fields. For instance, within MATLAB all vectors are assumed to be row vectors unless otherwise specified. To provide the developer more flexibility, this characteristic of the vector can be set within the pattern. If the keyword `ROW` is present within the property field, the vector is a row vector, and similarly, if the keyword `COLUMN` is present within the property field, the vector is a column vector. Also, although the index set can be used to set the rank of the variable to scalar, vector, or matrix, it is not possible to choose one of the other dynamic ranks. If the keyword `NOTSCALAR` is in the property field, the rank is set to NOT_SCALAR and if the keyword `NOTARRAY` is in the property field, then the rank is set to NOT_MATRIX. A final option is to define a variable that will match all ranks by placing the keyword `ANY` within the property field.

If a variable declared in the REPLACE block does not have a declaration in the WITH block, then the original declaration also holds for the WITH block. However, if a second declaration is present, then the properties of the variable will be modified to correspond to the new declaration. As a result, the variable declaration portion of the WITH block is optional.

In addition to declaring regular variables, user-provided routines can also be declared in the declaration portion of the WITH block for subsequent use in the statement portion of the block. These declarations are used when the algorithm developer wishes to replace MATLAB operations in the program with calls to specific routines. The declaration defines both the type of the routine call and the properties of the parameters, with the declaration format for both subroutine and function calls being provided in Figure 8.

```
FUNCTION function name ( argument list ){dim1, dim2}
SUBROUTINE function name ( argument list )
```

Figure 8: Declaration of user-provided routines.

Within the routine declaration, the *argument list* is responsible for indicating the order of the input and output arguments, since the use of the routine in the MATLAB code will keep the arguments in two separate lists. For instance, consider a Fortran subroutine `find_range` that inputs a list of numbers, `values`, and returns the minimum and maximum values, `min` and `max`, with the arguments in that order:

```
find_range( values, min, max ).
```

In MATLAB, this call would be:

```
[ min, max ] = find_range( values ).
```

To generate the correct output code for using this call, it is necessary to indicate the order of the arguments as one input argument followed by the two output arguments. To accomplish this, input arguments are indicated by `P_IN` and output arguments are listed by `P_OUT`. Furthermore, to allow the output variables to be declared of the correct size dynamically, it is necessary to define the dimensions of each output variable, using the format {*dim1, dim2*}, after each variable. The declaration of the `find_range` subroutine would be as follows:

```
SUBROUTINE find_range( P_IN, P_OUT{1,1}, P_OUT{1,1} ),
```

with `min` and `max` being declared as scalars. Note that the same format for declaring the size of the output variable is also used for declaring the return value of a function at the end of the `FUNCTION` statement.

If the output arguments are not a constant size, however, additional information is needed to define the size. Since the size of an output argument is typically based on the size of an input argument, it is also possible to label an input argument and use its dimensions to denote the size of the output variables. First, an input argument is labeled by placing an identifier after the input flag, `P_IN`{*id*}. Next, a special function, `DIM(` *id, dim_val* `)`, is provided to take the first or second dimension, with *dim_val* equaling `1` or `2` respectively, of the input argument labeled with *id*. For example, the following argument list would declare the output argument to be the same size as the input argument:

P_IN{a}, P_OUT{DIM(a,1), DIM(a,2)}.

A case that cannot be handled by the previous strategy is when the routine argument is used for both input and output. To handle this case, the MATLAB call is written using two arguments, with the input argument placed within the MATLAB call and the output argument placed on the left-hand side of the assignment. A third argument declaration type, P_IN_OUT, is then used in the declaration of the routine. There is no need to define the size of the output argument as it is assumed to be the same size as the input argument. The argument can be labeled, using the same format as the regular input arguments, so that its size can be used to define the size of other output arguments.

### 5.2.3 Transformation Statements

Within both the REPLACE and WITH blocks, following the variable declaration (if any), is a sequence of transformation statements that are either MATLAB statements or new statements that have been added to the transformation language. The MATLAB statements supported within the transformation language are the same statements that are supported by the FALCON environment for the input code.

New statements were added to the transformation language to provide the developer with the means for matching code fragments in a more abstract manner and to provide additional methods for code modification. The three primary statements are the BODY, a WHERE block, and an IN block.

The BODY statement is used to match any sequence of statements between two statements in the REPLACE block, with the format of the BODY statement being presented in Figure 9. Note that the BODY is labeled with an identifier (called *id*) in the example. This allows multiple BODY statements to be used within a transformation pattern.

*statement*$_1$
<BODY>(*id*)
*statement*$_2$

Figure 9: Format of the BODY statement.

The WHERE block allows dependence information to be used as part of the criteria for selecting the source code fragment in the WITH block. The two forms of the WHERE block are presented in Figure 10. The dependence list contains one or more dependence statements of the following form:

$$id_1 \;\; operator \;\; id_2.$$

An *id* in a dependence statement is either a variable within the program or the identifier of a BODY. There are four possible dependence operators: DOESNT_READ, DOESNT_WRITE, READS, and WRITES.

| | |
|---|---|
| *MATLAB assignment statement* | <BODY>(*id*) |
| <WHERE> | <WHERE> |
| *dependence list* | *dependence list* |
| <END> | <END> |

Figure 10: The two forms of the WHERE block.

The IN block is used to modify the statements matched by a BODY, with the format of an IN block being presented in Figure 11. Within the IN block, only a single MATLAB expression can be used for $MATLAB\ expression_1$ and $MATLAB\ expression_2$.

```
IN
        <BODY>(id)
    REPLACE
        MATLAB expression1
    WITH
        MATLAB expression2
    END
```

Figure 11: Format of an IN block.

### 5.2.4 Compiler and Algebraic Transformations

The IRS system is flexible enough to allow some traditional restructuring compiler transformations, such as loop unrolling and fusion, as well as the *algebraic transformations* of the matrix operations being performed to restructure the expressions that distinguish FALCON from other systems.

Of course, the restructuring of vector and matrix expressions using algebraic properties may sometimes have the same effect as the application of traditional restructuring techniques. However, during the interactive application, the algebraic transformations often provide a more convenient form for the user than the traditional loop-based techniques, even when the results are the same. For example, the conversion of an algebraic algorithm from a row-sweep to a column-sweep algorithm can typically also be represented as a loop-interchange. It is the support for such an overlap that allows the system to be molded to meet the developer's needs rather than forcing the developer to use a single type of transformation.

The typical scenario that demonstrates the range of transformations needed to support the development of codes revolves around the incremental increase in the amount of detailed information available about the effect and structure of the operators and operands involved. The IRS supports the application of transformations, propagation of structural and type information, and targeted code generation. The evolution of the code usually involves the application of various generalizations of some basic activities that include: specification of operand type, specification of operator algebraic and structural properties, operation customization, and operand representation selection. The operation customization is at the heart of the efficient code generation activity and is performed in response to the information provided by the other activities. For example, if the specification of the operators' algebraic and structural properties indicates that a particular matrix is in fact an orthogonal rank-1 perturbation to the identity, e.g., an Householder reflector, then a more efficient application of the matrix to a vector can be derived. The production of the more efficient version of the operation tends to involve decisions concerning: reordering of matrix and vector operations, substitution of equivalent operations, and the exploitation of specific operand structure such as sparsity or Toeplitzness.

In addition to reordering the operations after structure is specified, improving the efficiency of the computation of an expression can also be accomplished by replacing operations with equivalent operations. Some replacements can be effective for improving the performance, such as replacing expensive MATLAB operations with less costly operations, and other replacements may be used to improve the numerical stability of the code. As with most transformations, care must be taken to not destroy the numerical stability when the operations being performed are modified.

One class of these transformations is the decomposition of a matrix operation into lower-level operations. For instance, the following matrix-matrix multiplication operation,

$$A = B * C,$$

can be decomposed into multiple matrix-matrix operations, into matrix-vector operations, and into vector-vector operations:

$$\begin{aligned} A(i:l, j:k) &= B(i:l, :) * C(:, j:k), \\ A(:, j) &= B * C(:, j), \text{ and} \\ A(i, j) &= B(i, :) * C(:, j). \end{aligned}$$

Such decompositions could be used to generate code for parallel machines.

Conversely, multiple operations can be merged into a single operation. The best known example of this is the combination of multiple rank-1 updates to a matrix into a single rank-k update that is the basic algebraic transformation used when creating block methods from non-block methods to factor a dense matrix. For certain cases algebraic information must be used to create this block transform since the code for the rank-1 based version does not contain enough information to specify the block transform completely.

Another example of the exploitation of more subtle structure concerns the introduction of low-rank displacement representations of Toeplitz-like matrices in algorithms originally developed for dense matrices. In such a case, all operations with the Toeplitz-like matrix such as matrix-vector multiplication or the application of the transformation in a single step of a factorization algorithm is replaced by an associated operation on the *generator* of the matrix that results from a particular choice of displacement operator, i.e., an alternate and highly compressed representation of the Toeplitz-like matrix.

Another type of transformations, code generation transformations, are used to translate a MATLAB statement, or statements, into a subroutine or function call in the output language. The primary use of these transformations is to target operations provided by a specific library. This capability is important for several reasons, such as targeting a high-performance library for a particular machine or enabling multiple MATLAB operations to be combined into a single operation in the target library.

For example, consider the MATLAB operation of multiplying a transposed matrix times a vector,

$$a = B' * c.$$

The basic formulation of this in the AST would result in the generation of the transposed matrix followed by the actual multiplication,

$$\begin{aligned} Bt &= B' \\ a &= Bt * c. \end{aligned}$$

However, the matrix-vector subroutine provided by the BLAS supports this operation without requiring the transpose of the matrix. The pattern to perform the transformation the transposed matrix-vector operation is shown in Figure 12. Using this transformation, the time to perform an iterative solve using the QMR method, implemented in C++ and compiled using Gnu C++, was reduced from 30.29 seconds on a SPARCstation 10 to 8.73 seconds, a reduction of 71%.

For further details on the interactive restructuring system and other examples of its use, please refer to [MGG98].

```
REPLACE
    INTEGER n
    REAL A(n,n), b(n), c(n)

    c = A' * b;
WITH
    INTEGER i
    SUBROUTINE dgemv(P_IN, P_IN, P_IN, ...
       P_IN, P_IN{a}, P_IN, P_IN{b}, ...
       P_IN, P_IN, P_OUT{1, DIM(a, 1)}, P_IN)

    c = dgemv('C', DIM(A,1), DIM(A,2), ...
       1.D0, A, DIM(a,1), b, 1, 0.D0, 1);
END
```

Figure 12: Pattern for optimizing transposed matrix-vector multiplication.

# 6 Code Generation Phase

The Code Generation System (CGS) is responsible for converting the internal format, the AST, into the output format, one of the three target languages: Fortran 90, C++, and MATLAB's array language. In addition to the code generators, other important components of the CGS are the libraries of numerical operations that are required to support the functionality of the MATLAB language, including the class libraries for the C++ code.

## 6.1 Target Libraries

Non-intrinsic MATLAB functions (M-files) are in-lined by the Program Analysis system and hence present no challenge. Therefore, the MATLAB code generator does not need any support libraries outside of the MATLAB environment. FALCON supports the wealth of functionality of MATLAB's intrinsic functions by providing the code generators with translation mechanisms targeting a selected library. As implemented, the code generators provide a one-to-one translation between the function calls in the AST and the support libraries. One aspect of the IRS is the means to select among multiple subroutines that support the same functionality or to allow different libraries to be targeted. When coupled with the transformation system, this effectively provides the capability of true portable programming across a wide range of machines.

## 6.2 Fortran 90 Code Generator

The Fortran 90 code generator was the first to be implemented within the FALCON environment. It is simpler than the C++ code generator in two ways; the Fortran 90 language provides more built-in features for array operations, allowing many MATLAB operations to be translated directly into Fortran 90, and because it has the same data layout and subroutine interface as the Fortran 77 target libraries. The code generator, as written by DeRose, provides a one-to-one translation from the AST to Fortran 90.

The work of the code generator can be broken down into two basic functions, generating declaration statements for the variables and translating expression trees in the AST into Fortran 90 statements. When generating declarations, the code generator must first determine if the size of the variable is known or unknown. When the size is unknown, the variable will be declared as

allocatable and the shadow variables for the dimensions must be declared. If the type is unknown, then both a REAL and COMPLEX version of the variable will be declared and the shadow variable for the type must be declared. The code generator must also resolve the name of each variable instance with respect to both the SSA instances and to the *case insensitive* nature of Fortran. If the analysis, using the SSA method, has determined that multiple versions of a variable should be used (perhaps due to it having multiple types or ranks) an index number is appended to the variable name to differentiate the instances. A second problem arises from MATLAB recognizing variable names based on the case of the letters, while Fortran ignores the case. For instance, in MATLAB, the variables `A` and `a` are considered two separate variables while in Fortran they would be considered the same variable. If it is determined that two variable names are identical except for the case of the letters, then a flag is appended to one of the names to differentiate it.

When translating the expression trees into Fortran statements, the code generator utilizes the built-in Fortran 90 operators and functions for control structures, to perform scalar operations, and to perform the simpler operations on arrays and vectors such as addition and subtraction and for doing element-wise multiplication and division. The BLAS libraries are used for vector and matrix multiplication and the other target libraries are used to perform the higher-level operations, such as solves, as described in the previous section.

The translation of the dynamic memory allocation expressions is complicated by the manner in which Fortran 90 performs the dynamic data allocation within the scope of the current routine and frees the data upon its exit. Two attempts at writing a subroutine to perform dynamic memory allocations failed. In the first attempt, the compiler would not pass the unallocated matrix into the subroutine and, in the second attempt, the allocated data was freed when the subroutine returned. Instead of using a subroutine, the code generator inserts the dynamic allocation code for every allocation occurrence, increasing the size of the output program and making it more difficult to read, maintain, and modify the program.

The other code generators build upon the work done for the Fortran 90 code generator. However, they have been modified both to take advantage of language features present in their target language but not available in Fortran 90 and to provide code to support features present in Fortran 90 that are not present in their target language.

### 6.3 C++ Code Generator

The generation of the C++ code might have been accomplished following the same approach DeRose took with Fortran 90. The Fortran 90 Code Generator traverses the internal representation of the program in lexicographic order, outputting into a file the correct source code for each statement and expression as it is encountered. This allows the file containing the Fortran 90 program to subsequently be compiled by Fortran 90 compilers and restructurers. An alternative approach, however, is to directly translate the internal format of FALCON into the internal format of another restructuring tool. This allows subsequent restructuring to occur without the need to first generate the source code that would then need to be parsed. In turn, the restructuring tool would be responsible for either generating the source code or performing the compilation. With the availability of the Sage++ library [BBG+93], a class library for building Fortran 90 and C++ restructuring tools, it was decided to take this second approach for the C++ code generation.

Whereas the Fortran 90 code is generated in a single step, this alternative approach requires that the C++ code be generated in two steps. First, the program is translated from the internal format of FALCON into the internal format of Sage++ and, second, the Sage++ tools are used to generate the source code. Once the code has been translated into the Sage++ format, however, additional optimizations could be applied before the source code is generated.

### 6.4 Translation from FALCON to Sage++

A C++ Code Generator was developed to translate the FALCON internal format, the AST, into the internal format of Sage++. Though this code generator is based upon the Fortran 90 code generator developed by DeRose, the differences in the target languages coupled with the different approaches to the code generation necessitated substantial modifications.

As previously mentioned, the Fortran 90 code generator traverses the AST in lexicographic order, generating the source code for each statement, expression, symbol, and value as it is encountered. The C++ code generator, on the other hand, builds a Sage++ object to represent the entire program. To build the program, symbols and values are combined to form expressions, statements are built from these expressions, and statements are combined into other statements. Since the AST was developed in conjunction with the Fortran 90 code generator, it closely represents the semantics of Fortran 90 and, as a result, the code generator needs to understand very little of the semantics in order to generate the correct code. The ordering of the nodes in the tree inherently contains this information. In contrast, the C++ code generator must examine the information present in the AST to determine the correct operation to be performed and, based upon the semantics of that operation, create the correct C++ object.

For example, the generation of the code to represent the MATLAB loop constructs is straightforward in the Fortran 90 code generator and more complex in the C++ code generator. Consider the MATLAB `for` loop construct,

`for` *loop index* `=` *lower bound*`:`*stride*`:`*upper bound*`,`

and AST representation of this construct in Figure 13. By performing an in-order traversal of the

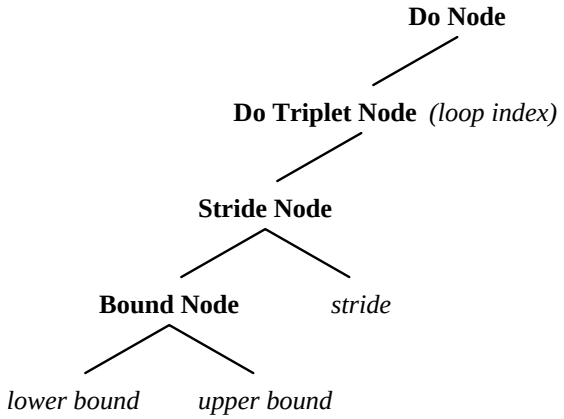


Figure 13: AST representation of the `for` loop construct.

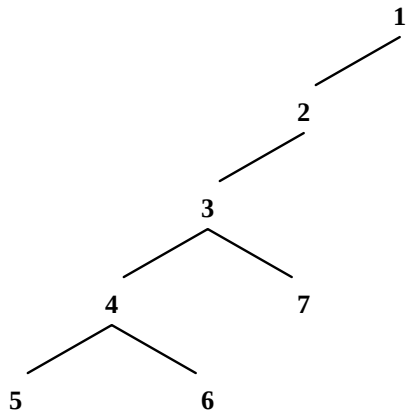


Figure 14: In-order traversal of the `for` loop construct.

nodes in the AST, as shown in Figure 14, the Fortran 90 code generator is able to generate the correct code by performing the operations at each node as shown in Table 1. Note that the only information required to perform the operation at each node is the information present in that node. In contrast, to perform the operations required by the C++ code generator, as shown in Table 2, information must be shared between the nodes. This sharing of information is necessary not only to build the expression objects, but also to select the conditional operator based on the sign of the *stride*.

Whereas Fortran 90 provides built-in matrix operations, these same operations are provided in C++ through the use of a matrix class. Though the class structure can support most of the

| Node | Operation |
|---|---|
| 1 | Output `DO` |
| 2 | Output *loop index* `=` |
| 5 | Output *lower bound* |
| 6 | Output *upper bound* |
| 7 | Output *stride* |

Table 1: Translation of the `for` loop to Fortran 90.

| Node | Operation |
|---|---|
| 5 | Build expression object for *loop index* = *lower bound* |
| 6 | If *stride* is positive then<br>Build expression object for *loop index* $\leq$ *upper bound*<br>Else<br>Build expression object for *loop index* $\geq$ *upper bound* |
| 7 | Build expression object for *loop index* += *stride* |
| 1 | Build loop object using *lower bound* expression, *upper bound* expression, and *stride* expression. |

Table 2: Translation of the `for` loop to C++.

necessary operations, a few of the constructs are difficult to reproduce in this manner. As a result, modifications to the code generator were performed to support a few of the matrix operations.

For instance, consider the vector constructor from MATLAB, which may contain symbols, vector expressions, and vector sections:

```
v1 = [ v2 a:b v3(c:d) ].
```

Within Fortran 90, this can be represented as a single statement through the use of an *array constructor*. Using the same example, the generated Fortran 90 code would be:

```
v1(1,:) = (/ v2, (/ (a + T__12, T__12 = 0,int((b-a))) /), v3(c:d) /).
```

Since this construct is not provided by C++, a different approach is required in the C++ code generator.

One alternative is to first construct a single vector that represents all the elements on the right-hand side and then assign this vector to the left-hand side. A second alternative is to individually assign each expression on the right-hand side to the appropriate section of the vector on the left-hand side. The second approach was chosen for the C++ code generator since it avoids the need to allocate a temporary array, even though this approach may require additional statements.

Additional modifications to the C++ code generator were required based upon differences in the implementation of the matrix operations within the matrix class as compared to Fortran 90. Before these modifications are explored, however, the matrix class will first be discussed.

### 6.5 Development of a Matrix Class

With the decision to support the generation of C++ code from the FALCON environment, it was known that a matrix class, or classes, would be required to provide the necessary matrix operations. Given that a number of matrix classes are freely available, the long term goal is to have the FALCON

environment support multiple matrix classes and allow the developer to select the class. Initially, however, it was decided to support only a single matrix class.

The next decision to be made was whether to utilize an existing matrix class with the FALCON environment or to develop a new matrix class. One problem with the use of an existing matrix class is the difficulty in determining if all the necessary functions are provided. Of special concern is the support for all the MATLAB array reference types, including the use of a vector expression with a stride as an index:

*v(lower:stride:upper).*

For instance, according to their documentation, neither the C++ Matrix Class from Tisdale [Tis93], the MatClass from Birchenhall [Bir93], nor newmat07 from Davies [Dav94] support this type of reference. A matrix class that did appear to meet most of the computational needs of the system, including support for array indexing with a stride, was the LAPACK++ matrix class from Dongarra, Pozo, and Walker [DPW93].

It was decided, however, that by developing a matrix class in conjunction with the C++ code generator, it could both be tailored to meet the needs of the FALCON environment and to allow the exploration of other issues, such as the improvement of the allocation and reallocation of data. The development of the matrix class also required decisions to be made with respect to functionality and performance.

### 6.5.1 Supported Data Types

Given the capabilities of the C++ classes, it is possible to develop a single class that supports all of the MATLAB data structures, which is the approach taken by the MATCOM compiler [Yar96]. This approach, however, would lose the rank and type information that had been found by the inference techniques and would require significantly more run-time decisions. It was decided, therefore, to have multiple types that support scalars, vectors, and matrices for both real and complex values. The data types that were developed are as follows:

**Real Scalars** – Real scalars are implemented using the `double` type defined within the language for double precision variables. In certain situations, a scalar integer, `int`, will also be used.

**Complex Scalars** – Complex scalar values are implemented using the `double_complex` class provided in the standard library.

**Real Vectors** – This class was developed to perform operations on vectors of double precision elements when the length of the vector is known.

**Complex Vectors** – This class was developed to perform operations on vectors of double precision complex elements when the length of the vector is known.

**Real Matrices** – This class was developed to perform operations on 2-dimensional matrices of double precision elements. In addition, this class is used to support real vectors of unknown size. This was done to replicate the approach taken with the generation of the Fortran 90 code.

**Complex Matrices** – This class was developed to perform operations on 2-dimensional matrices of double precision complex elements. In addition, this class is used to support complex vectors of unknown size. This was done to replicate the approach taken with the generation of the Fortran 90 code.

The basic functions of the matrix classes are the same, regardless of the data types, and can be divided into three areas: matrix allocation, referencing values, and numerical operations.

### 6.5.2 Matrix Allocation

Matrices in MATLAB are dynamically allocated when needed and allowed to grow as they are accessed. To support this type of functionality, two types of allocation methods are needed within the matrix classes. The first allocation method is used for allocating the entire matrix. This is accomplished in two steps, first any existing data space is deallocated and then the data space is allocated using the new size.

The second allocation method is that which is used for extending the matrix. This allocation is performed with three slightly different steps. First, the new data space is allocated, next the data in the old data space is copied into the new data space, and then the old data space is deallocated.

### 6.5.3 Matrix Reference Types

MATLAB provides four types of references that must be supported by the matrix class:

$A$ – Reference to the entire matrix.
$A(i, j)$ – Reference to a single element.
$A(l_i : u_i, l_j : u_j)$ – Reference to a contiguous sub-matrix.
$A(l_i : s_i : u_i, l_j : s_j : u_j)$ – Reference to sub-matrix, with strides between the elements.

However, only the first two reference types are supported by the C++ language. Therefore, the matrix class must contain methods to support the referencing of sub-matrices.

One possibility for implementing the sub-matrix references is to restrict their use to simple assignment statements where the right hand side consists of a single variable reference. With this approach, only assignment statements need to support references to sub-matrices. This approach, however, would require more temporary variables and additional copying of data.

A second approach is to create a *reference* to the matrix that incorporates the additional information regarding the requested sub-matrix. During the subsequent use of the reference, this information is utilized to correctly access the necessary elements. This is the approach that was taken by other classes, such as LAPACK++ and the C++ Matrix class, and can reduce the need to copy the data. As the colon operator is not easily supported by the C++ language, the different matrix references are identified by the number of parameters in the reference. For instance, a matrix reference with two parameters refers to a single element, a reference with four parameters refers to a contiguous sub-matrix, and a reference with six parameters refers to a sub-matrix that has strides between the elements. The reference types for both dimensions must be the same, i.e. it is not possible to use a single value for the first dimension, $i$, and a range of values for the second dimension, $j : k$. The reference with the three parameters would be $A(i, j, k)$ and the matrix class would not be able to differentiate $A(i : j, k)$ from $A(i, j : k)$; instead, the reference would need to be $A(i, i, j, k)$.

### 6.5.4 Numerical Operations with Matrices

All the numerical operations provided by MATLAB should be supported by the matrix classes. Currently, the matrix classes support arithmetic operations such as addition, subtraction, multiplication, and division, in addition to trigonometric functions such as cosine, sine, and tangent. The logical operations supported by the matrix classes include testing whether elements are equal or not equal. Other functions for finding maximum and minimum values are also supported. The classes

support these operations not only between two variables from the same class, but also between variables from different classes.

Where possible, the numerical operations within the matrix classes are implemented using publicly available libraries, such as the BLAS. In order to facilitate the use of such libraries, the actual data elements are stored in column-major order, which is the format used by Fortran. This allows the C++ code to make use of the same optimized routines which the Fortran 90 code utilizes. Before the data can be passed to one of these routines, however, the matrix reference must first be checked to see if it refers to a sub-matrix with strides between the elements. As some of the routines require that data elements be located in contiguous storage, the sub-matrix elements may need to be copied into a contiguous temporary space before being passed to the routine.

# 7 Experimental Results

FALCON's performance was evaluated in comparison to MATLAB and other execution methods for MATLAB files. In the remainder we present the experimental procedures, results and conclusions.

Table 3 shows the five execution methods that were used for the performance evaluation. These tests utilized three different MATLAB compilers and generated code in three different languages in addition to the MATLAB language.

| Method | Code | Compiler | Libraries |
|---|---|---|---|
| FALCON | Fortran 90 | Vast Fortran 90<br>Sun f77 (-O3) | BLAS, LINPACK, LAPACK,<br>and EISPACK |
| FALCON | C++ | GNU C++ (-O) | BLAS, LINPACK, LAPACK,<br>and EISPACK |
| Mathworks<br>mcc | C-MEX | GNU C (-O3) | Provided<br>within MATLAB |
| MATLAB | M-file | | Provided<br>within MATLAB |
| MATCOM | C++ | GNU C++ (-O) | BLAS and those<br>provided by MATCOM |

Table 3: Execution methods.

To compare the methods, twelve MATLAB programs were collected, as shown in Table 4. All but the three iterative linear system solvers (CG, QMR, and SOR) require element-wise access of the matrices.

All performance tests were conducted on a SPARCstation 10 running in multi-user mode. Each program was run five times for each execution method, with the best CPU execution time being selected. The results from these tests are presented in Table 5. As the execution methods utilize similar underlying computational functions, such as the BLAS and LINPACK, the primary differences in execution time come from the overhead incurred by the different methods and differences in the program analysis. The performance of the FALCON-generated code was compared to each of the other three execution methods.

## 7.1 Comparison Between the FALCON Generated Codes

The Fortran 90 execution times are faster for ten of the twelve programs, with the biggest speedups occurring for the programs with element-wise computations. Of the two programs where the C++

| Test Programs: | | Problem size | Source |
|---|---|---|---|
| Adaptive Quadrature Using Simpson's Rule | (AQ) | 1 Dim. (7) | a |
| Preconditioned Conjugate Gradient method | (CG) | $420 \times 420^*$ | b |
| Crank-Nicholson solution to the heat equation | (CRN) | $321 \times 321$ | a |
| Dirichlet solution to Laplace's equation | (DIR) | $41 \times 41$ | a |
| Finite Difference solution to the wave equation | (FDF) | $451 \times 451$ | a |
| Galerkin method to solve the Poisson equation | (GAL) | $40 \times 40$ | c |
| Incomplete Cholesky Factorization | (ICN) | $400 \times 400$ | d |
| Two body problem using Euler-Cromer method | (OEC) | 6240 steps | c |
| Two body problem using 4th order Runge-Kutta | (ORK) | 3200 steps | c |
| Quasi-Minimal Residual method | (QMR) | $420 \times 420^*$ | b |
| Successive Overrelaxation method | (SOR) | $420 \times 420^*$ | b |
| Generation of a 3D-Surface | (3D) | $51 \times 31 \times 21$ | d |

Source:
a: [Mat92b] b: [BBC+93] c: [Gar94] d: Colleagues

* A stiffness matrix from the Harwell-Boeing Test Set (BCSSTK06) was used as input data for these programs.

Table 4: Test programs.

code is faster, the AQ program illustrates the advantage of the improved memory allocation strategy. The differences in performance between the Fortran 90 and C++ are solely related to the overheads incurred by the respective language.

To determine why the C++ code is usually slower than the Fortran 90 code, three of the programs, CRN, OEC, and QMR, were explored in more detail.

Each loop iteration of the CRN program requires the solution of a tridiagonal linear system. Each of these vector accesses is performed by the invocation of a method from the vector class. However, in order to support references to sub-arrays defined with strides, the access method is required to test for a stride during each access. To determine the amount of overhead the accesses are incurring, the C++ code was modified, by hand, to replace the access method with a reference to the values. When the modified code was run, the execution time dropped to 0.67 seconds, a reduction of 71%. In turn, the slowdown when compared to the Fortran 90 code, was reduced from a factor of 7.1 to a factor of 2.1.

In each loop iteration of the OEC program, several calculations are performed using vectors of length two. As a result, the overhead required to invoke methods to perform the calculations is high relative to the amount of time that is spent doing the numerical operations. As two of the operations are vector-vector updates, $a = a + \alpha b$, a transformation pattern was written to replace the updates with a direct call to the BLAS primitive. When this transformation was applied, the execution time dropped to 0.63, a reduction of 44%, and the slowdown, when compared to the Fortran 90 code, was reduced from a factor of 6.3 to a factor of 3.5.

Upon examination of the QMR program, it was determined that the Fortran 90 code generator was utilizing the BLAS `dgemv` primitive to multiply the transposed matrix times the vector, while the C++ code was first transposing the matrix and then performing the multiplication. A transformation pattern for the purpose of recognizing the multiplication by the transposed matrix was generated and applied to the C++ code. With the improved C++ code, the execution time dropped to 8.73 seconds, a reduction of 71%. This reduced the slowdown, when compared to the Fortran 90 code, from a factor of 3.8 to a factor of 1.1.

| Program | FALCON | | MATLAB | | MATCOM |
|---|---|---|---|---|---|
| | C++ | Fortran 90 | C-MEX File | M-File | C++ |
| AQ | 20.30 | 25.71 | 25.70 | 59.47 | † |
| CG | 15.00 | 14.24 | 58.42 | 46.98 | 44.91 |
| CRN | 2.27 | 0.32 | 1.83 | 61.27 | 63.26 |
| DIR | 1.38 | 0.27 | 4.42 | 75.63 | 52.35 |
| FDF | 1.43 | 0.28 | 1.28 | 49.23 | 52.41 |
| GAL | 1.47 | 0.73 | 1.30 | 55.17 | 33.77 |
| ICN | 3.67 | 1.59 | 5.08 | 53.90 | 31.25 |
| OEC | 1.13 | 0.18 | 17.37 | 72.62 | 23.08 |
| ORK | 1.90 | 0.22 | 39.53 | 53.48 | 25.61 |
| QMR | 30.29 | 8.06 | 60.93 | 55.30 | 87.61 |
| SOR | 10.35 | 21.86 | 66.08 | 62.25 | 116.70 |
| 3D | 16.18 | 10.47 | 29.42 | 65.12 | ‡ |

† - A segmentation fault occurs when executing the code.
‡ - The code generated by MATCOM contains an infinite loop.

Table 5: Execution time, in seconds, for the execution methods.

After these tests, it is clear that the overhead to invoke the C++ methods for small data accesses incurs a significant performance penalty. As Fortran 90 has these type of operations built into the language, it has noticeable performance advantage for such problems.

### 7.2 Comparison with MATLAB C-MEX

When compared to the MATLAB C-MEX codes, the FALCON Fortran 90 code is faster for the eleven of the twelve program, with the speedups ranging from 1.8 to 179.7, and the same speed for the twelfth. The FALCON C++ execution times are faster for nine of the twelve programs, with the speedups ranging from 1.3 to 20.9. Of the three programs where the CMEX execution was faster, the C++ program was slower by 12% to 24%. Our analysis determined that CMEX programs have better support for element-wise array accesses than the generated C++ code. In contrast, when accesses are made to short vectors, as in the OEC program, the C++ program performs much better than the CMEX program, with the C++ program being 15.4 times faster. For instance, the `mcc` compiler used to generate the CMEX file implements both of the vector-vector updates, $a = a + \alpha b$, as matrix-matrix updates using doubly-nested loops. Also, to calculate the norm of the vector, the CMEX file packages the vector and passes it back to a MATLAB library to perform the operation. Once the operation is performed, the results must be imported back into the CMEX file. In summary, the FALCON C++ code is generally faster than the CMEX code, with the FALCON C++ code having the most difficulties with programs that use element-wise access. We also note that the FALCON program analysis system typically infers more information about the variable properties than the `mcc` compiler; see [DeR96].

### 7.3 Comparison with MATLAB M-File

Our comparisons show also that the FALCON Fortran 90 and C++ codes are faster in all tests than the respective MATLAB M-Files, with speedups ranging from 2.3 to 403.4 for Fortran 90 and from 1.8 to 64.3 for C++. Some of these speedups could be further improved using the techniques

described above. Specifically for the moderate speedups observed for CG and SOR, we observe that most of their time is spent in BLAS primitives performing matrix-vector multiplications or triangular solves. Since both the C++ code and the M-File code call numerical libraries to perform these operations, the numerical operations will take a similar amount of time in this case. In this case, the speedup comes from reducing the loop overhead and improving the performance of the scalar and vector operations.

### 7.4 Comparison with MATCOM C++

When comparing the FALCON Fortran 90 code to the MATCOM code, the FALCON code is faster for all ten program where the MATCOM code executed successfully, with the speedups ranging from 3.2 to 197.7. The FALCON C++ execution times are also faster for all ten programs, with the speedups ranging from 2.9 to 37.9.

When comparing the two C++ execution methods, this comparison highlights the difference between the utilization of extensive program analysis to determine variable properties, the FALCON approach, and the use of a single powerful matrix class to perform these determinations at run time, the MATCOM approach. Not only does the FALCON C++ code execute faster than the MATCOM code for the ten programs, the compiled MATCOM code is slower than the MATLAB M-Files for four of the ten programs. This indicates that the run-time decisions of the MATCOM matrix class are sometimes even slower than the run-time decisions of the MATLAB interpreter.

For eight of the programs, the FALCON C++ code runs at least 8.5 times faster than the MATCOM code. These programs include those that operate on both entire matrices and element-wise accesses. Of the two remaining programs, if the FALCON version of the QMR program is improved using the `dgemv` optimization, its speedup increases to 10.0. This clearly shows that utilizing the program analysis system of FALCON to eliminate run-time decisions provides a distinct performance advantage.

## 8 Conclusions

We have described the FALCON environment that allows numerical algorithms to be developed with a more algebraic language, MATLAB, and then translated into traditional languages that provide better performance. It has been shown through experimentation that the code generated by FALCON is faster than the other approaches in almost all cases. More detailed experimental results can be found in [DeR96] and [Mar97]. It has also been successfully used in the development of more complex codes for synthetic aperture radar applications in cooperation with D. Munson and J.A. Lee of the University of Illinois. In addition, the flexibility and extensibility of FALCON lends itself to iterative nature of algorithm development and library support. As a result, FALCON fills a niche in the high-performance software hierarchy that has not received much attention in recent years. Tremendous effort has been spent on the development of specific sets of low-level high-performance computational and communication kernels for each new architecture that rose to prominence and the associated specific numerical libraries that could be built upon them. Extensive work has also been performed at the full PSE level where it is assumed that high-performance libraries for specific machines are available and most of the effort of the environment is spent on selecting solution approaches based on problem characteristics and managing input/output data in a fashion that is appropriate for application researchers and not computing specialists. Until recently, very little attention has been paid to environmental support in the middle of these two extremes, i.e., for the designers and implementers of the required high-performance libraries on multiple architectures and in multiple applications; or more computationally sophisticated applications researchers that

are interested in rapid prototyping and the assessment of the use of more advanced algorithms in their areas. FALCON is an attempt to provide such environmental support and has shown initial promise in delivering such capabilities.

There are, of course, many improvements and extensions that need to be pursued. Current work centers on the development of tool boxes that support specific algorithmic and algebraic transformations seen in the development of deterministic and stochastic iterative methods for sparse and structured problems problems. The initial work on integration of C++ and matrix classes into the code generation system indicates that extensions to support more standard off-the-shelf matrix classes (despite their limitations vis a vis MATLAB) should be considered along with the use of algebraic and application-specific information to restructure the encapsulation schemes used in a particular class. Further work is needed on customizing the interface of the codes produced in order to more efficiently embed it within an existing application code. A related activity is the demonstration that libraries developed under a system like FALCON could have new versions developed via algebraic transformations as new architectures became available thereby mitigating the massive efforts of the past to recast existing libraries without significant functional extension in response to moderate changes in an architecture. Of course, the new version of the library would still have to be tuned but a substantial reduction in the startup time of such an activity might be possible. In the limit, if such a demonstration was successful it might be possible to develop library code generators that create semi-custom versions of library codes at compile time. Rather than requiring the development of particular versions of each algorithm/implementation variant in a library, basic building blocks and a construction procedure would be created by the designers and a future version of a FALCON-like environment used for library compilation/customization.

## References


[BBC+93] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Publications, 1993.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, and S. Srinivas. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. Technical report, Indiana University, November 1993.

[Bir93] C. R. Birchenhall. *A Draft Guide to MatClass: A matrix class for C++*, 1993.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[CC93] James R. Cordy and Ian H. Carmichael. The TXL Programming Language Syntax and Informal Semantics Version 7. Technical Report 93-355, Department of Computing and Information Science, Queen's University, Kingston, Canada, June 1993.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

[CK92] Steve Carr and Ken Kennedy. Compiler Blockability of Numerical Algorithms. In *Proceedings, Supercomputing '92*, pages 114–124, November 1992.

[Dav94] Robert B. Davies. Writing a matrix package in C++. In *OON-SKI'94 Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 207–213, April 1994.

[DeR96] Luiz Antonio DeRose. *Compiler Techniques for MATLAB Programs.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[DGG+94] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. An environment for the rapid prototyping and development of numerical programs and libraries for scientific computation. In F. Makedon, editor, *Proc. of the DAGS'94 Symposium: Parallel Computation and Problem Solving Environments*, pages 11–25, Dartmouth College, July 1994.

[DPW93] Jack J. Dongarra, Roldan Pozo, and David W. Walker. *LAPACK++: A Design Overview of Object-Oriented Extensions for High-Performance Linear Algebra.* University of Tennessee, 1993.

[DT97] M. Dæhlen and A. Tveito. *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, Boston, 1997.

[Gar94] Alejandro L. Garcia. *Numerical Methods for Physics.* Prentice Hall, 1994.

[GHR92] E. Gallopoulos, E. N. Houstis, and J. R. Rice. *Future Research Directions in Problem Solving Environments for Computational Science.* Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science. Technical Report 1259, Center for Supercomputing Research and Development, Oct. 1992. Workshop held at NSF, Washington, D.C., Apr. 1991.

[GPS90] K. A. Gallivan, P. J. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32(1):54–135, March 1990.

[GS93] William Gropp and Barry Smith. Simplified Linear Equation Solvers Users Manual. Technical Report ANL-93/8-REV 1, Argonne National Laboratory, June 1993.

[JKR92] P. Jacobson, B. Kågström, and M. Rännar. Algorithm development for distributed memory multicomputers using CONLAB. *Scientific Programming*, 1:185–203, 1992.

[Mar97] Bret Andrew Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation.* PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[Mat92a] The Math Works, Inc. *MATLAB, High Performance Numeric Computation and Visualization Software. Reference Guide*, 1992.

[Mat92b] John H. Mathews. *Numerical Methods for Mathematics, Science and Engineering.* Prentice Hall, 2nd edition, 1992.

[Mat95] The MathWorks, Inc. *MATLAB Compiler*, 1995.

[MGG98] B. Marsolf, K. Gallivan, and E. Gallopoulos. The Interactive Restructuring of MATLAB Programs using the FALCON Environment. In Alex Veidenbaum and Kazuki Joe, editors, *Innovative Architecture for Future Generation High-Performance Processors and*

*Systems*, pages 3–12. IEEE Computer Society Press, 1998. Proceedings for the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems-IWIA'97, Maui, Hawaii, October 22-24, 1997.

[Sch75] J. T. Schwartz. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM*, 18:722–727, 1975.

[Tis93] E. Robert Tisdale. *C++ Matrix Class*. UCLA, 1993.

[TP93] Peng Tu and David Padua. Automatic Array Privatization. In U. Banerjee, D. Gelemter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 500–521. Springer-Verlag, August 1993. 6th International Workshop, Portland, Oregon.

[Wea94] Stephen Andrew Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Master's thesis, University of Illinois at Urbana-Champaign, 1994.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, 1996.

[Yar96] Keren Yaron. *MATCOM, A MATLAB to C++ Translator*, March 1996.