AspectMatlab++: Annotations, Types, and Aspects for Scientists

Andrew Bodzay
School of Computer Science, McGill University,
Montreal, QC, Canada
andrew.bodzay@mail.mcgill.ca

Laurie Hendren
School of Computer Science, McGill University,
Montreal, QC, Canada
hendren@cs.mcgill.ca

Abstract

In this paper we present extensions to an aspect oriented compiler developed for MATLAB. These extensions are intended to support important functionality for scientists, and include pattern matching on annotations, and types of variables, as well as new manners of exposing context. We provide use-cases of these features in the form of several general-use aspects which focus on solving issues that arise from use of dynamically-typed languages. We also detail performance enhancements to the ASPECTMATLAB compiler which result in an order of magnitude in performance gains.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Processors*]: Compilers

General Terms Languages

Keywords Aspect-oriented programming, MATLAB, ASPECTMATLAB compiler

1. Introduction

MATLAB [8–10] is a dynamic array-based programming language, which has widespread use throughout the scientific community. Its success stems from its convenience as a dynamic scripting language - providing the programmer with high-level matrix operators, a large library of built-in functions, a flexible syntax which requires no type declarations, as well as a quick and easy development through the MATLAB IDE. These factors all make the language very appealing to novice and expert programmers alike, as development requires little training, and code can be rapidly prototyped.

Our work builds upon the ASPECTMATLAB project, which introduced the idea of aspects in MATLAB [1, 2]. The original focus of ASPECTMATLAB was that prominent features in scientific programming, such as loops and arrays could be easily matched and operated upon. Our challenge was to further develop aspect-oriented programming for MATLAB, in a way that is consistent with the ease of use of the base language. We wanted to introduce language mechanisms through which programmers could more easily understand and control their code, with the goal of preventing errors as well as improving performance. Simultaneously, we aim to make ASPECTMATLAB more accessible to current MATLAB users.

Unlike more formal programming languages, MATLAB has neither static type declarations nor static type checking, with type information being determined at run-time. While this feature makes prototyping quick, it can also result in type inconsistencies which can propagate throughout a program. One of our aims is to accommodate scientists by creating language extensions and scientific aspects which help them to understand and deal with these sorts of typing issues that arise in environments with no static types. In the same vein, we also seek to help scientists reason about atypical forms of types, such as units, that may occur throughout their computations.

To achieve these goals, we implement several new language features. These include a number of new patterns, such as the `annotate` pattern, which allows pattern matching on specially formatted MATLAB comments. We also introduce `type` and `dimension` patterns, which match join points corresponding to a particular MATLAB base type or array size respectively. In addition, we now allow for context exposure of the loop body in all patterns which match loops. This enables ASPECTMATLAB programmers to implement aspects which rewrite and modify the loop body.

Another important challenge was to improve the performance of ASPECTMATLAB code. By inserting aspect code directly into the base MATLAB code, as well as making local copies of aspect properties we eliminate a significant amount of overhead.

The main contributions of this paper are:

Extensions to ASPECTMATLAB language: Building upon the ASPECTMATLAB language, we introduce a number of language extensions which improve the ease of use. These include the introduction of the annotate, type, and dimension patterns, as well as context exposure of loop body in loop patterns.

Scientific aspects: We have developed a library of new aspects which provide easy-to-use new functionality for ASPECTMATLAB users such as on-the-fly type and unit checking.

Improvements to the ASPECTMATLAB compiler: Several functions of the ASPECTMATLAB compiler have been reworked, in order to support not only new language features, but to improve general performance

Experimental evaluation: We have tested the ASPECTMATLAB language on a set of benchmarks to demonstrate the feasibility of woven code as well as the importance of our weaving optimizations. With the optimizations we have included, aspects are shown to have an acceptable amount of overhead.

The remainder of this paper is organized as follows. In Section 2 we provide the key background of MATLAB and ASPECTMATLAB. Section 3 describes the ASPECTMATLAB language extensions and Section 4 outlines the aspects we have included in our library. In Section 5 we discuss key improvements we made to the

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA
Copyright 2015 ACM 978-1-4503-3249-1/15/03...$15.00
http://dx.doi.org/10.1145/2724525.2724573

implementation of ASPECTMATLAB and in Section 6 we show the performance of aspect woven code. We finish with related work in Section 7 and conclusions and future work in Section 8.

2. Background

2.1 MATLAB Types Overview

MATLAB, unlike more many more formal programming languages, does not have any static type declarations or static type checking. The default type for storing information in MATLAB is a matrix, with even scalar variables being stored as a 1x1 array. Matrices have two important characteristics, their dimensions, representing the size of the matrix, and their base type, which represents the variety of data the matrix is allowed to hold. The base data type of a MATLAB variable can be one of several default types, such as the numeric types `double` or `int32`, or they can be user-defined.

In MATLAB, type information is determined at run-time. Due to the lack of type declarations, as the program progresses, the same variable may contain values of different types. While the lack of static type declarations makes for fast prototyping, MATLAB programmers often do have an idea as to what base types and dimensions their variables are expected to have throughout their program. Despite the fact that no formal type declarations exist, many MATLAB functions have comments which specify the types expected of its arguments. An example of this variety of informal declaration is shown in Figure 1. While this information is not leveraged by MATLAB itself, failure to meet these informal recommendations can result not only in run-time errors, but also incorrect results. Type errors of this variety can easily propagate throughout a program, making it difficult to determine from where the error originated.

```
function [F, V] = nbody3d(n, R, m, dT, T)
%-------------------------------%
% This function M-file simulates the gravitational
%   movement of a set of objects
% Invocation:
%   >> [F, V] = nbody3d(n, R, m, dT, T)
%   where
%   i. n is the number of objects,
%   i. R is the n x 3 matrix of radius vectors,
%   i. m is the n x 1 vector of object masses,
%   i. dT is the time increment of evolution,
%   i. T is the total time for evolution,
%
%   o. F is the n x 3 matrix of net forces,
%   o. V is the n x 3 matrix of velocities.
%-------------------------------%
```

Figure 1. Header of MATLAB simulation of n-body problem [14]

2.2 ASPECTMATLAB

In ASPECTMATLAB, aspect definitions were developed as an extension to MATLAB object-oriented class definitions. Object-oriented MATLAB classes are allowed to contain a `properties` block, where data that belongs to an instance of the class is defined. These properties can be defined with default values or initialized in the class constructor. Object-oriented MATLAB classes also allow for a `methods` block, which can include class constructors, property accessors, or ordinary MATLAB functions. ASPECTMATLAB expands upon this by allowing for two additional blocks: `patterns` and `actions`. Patterns, which are analogous to pointcuts in other aspect-oriented languages, are used to pick out sets of join points in program flow. Actions, which are analogous to advice, are blocks of code that are intended to be joined to specific points of the base program. Actions specify what should be done when code is matched by patterns.

In Figure 2 we see an example which makes use of these four features of aspects. The `properties` block defines a counter, which is initialized at its declaration and can be used throughout the aspect. The `methods` block defines a function called increment. In the `patterns` block, we define a pattern, called callAdd, that we want to match in the base code. In this case, we match calls to the function add. Finally, the `actions` block defines an action called actCall. This action specifies that we should call the method increment after every join point in the base code which matches the pattern callAdd. It then displays the name of the function.

```
aspect myAspect
properties
    counter = 0;
end

patterns
    callAdd : call(add);
end

methods
    function increment( this )
        this.counter = this.counter + 1;
    end
end

actions
    actCall : after callAdd : (name)
        this.increment ();
        disp([' calling ', name]);
    end
end

end
```

Figure 2. Simple ASPECTMATLAB example

ASPECTMATLAB was introduced with a variety of patterns to match basic language constructs. An emphasis was made on patterns which match array and loop constructs. It introduced several function matching patterns, `call`, `execution`, and `op` which match calls to a specified function, the execution of a specified function, and calls to basic operations respectively. These patterns all take as a parameter the name of the function or operation they should match. For example, in Figure 2, we see that the `call` pattern takes as a parameter 'add', meaning that it will match calls to the function add. ASPECTMATLAB has two array-related patterns `get` and `set`, which allow for matching accesses of and assignments to arrays respectively. Similarly to the function matching patterns, the `get` and `set` patterns take as a parameter the identifier of the variable they should match. ASPECTMATLAB also features loop-related patterns `loop`, `loophead`, `loopbody` which allow for matching on various portions of loops in MATLAB. Unlike functions and array accesses, there is no easy way to identify specific loop join points in the base code. Instead, loops are specified by the name of the loop iterator variable. Finally, the `within` pattern allows for limiting the context with which matches can be made to join points within a specified construct, such as a function or a loop.

In order to match more complex patterns, ASPECTMATLAB allows for compound patterns to be created using logical combinations of primitive patterns. An example of this is shown in Figure 3. `pCallTest` will match all calls to the function test that occur within a loop, `pGetOrSet` will match array access and assignments occurring within a function `add`, and `pCallExec` matches the pre-

viously defined pattern `pCallTest` as well as the execution of the test function itself.

```
patterns
  pCallTest : call( test ) & within(loops, *);
  pGetOrSet : (get(*) | set(*)) & within(function, add);
  pCallExec : pCallTest | execution( test );
end
```

Figure 3. Compound ASPECTMATLAB patterns

There are three types of actions in ASPECTMATLAB, `before`, `around`, and `after`, which specify when, in relation to a matched join point, a piece of code should should be executed. As one might expect, `before` actions are woven directly before a join point, and `after` actions are woven directly after a join point. `around` actions are different, in that they replace the join point completely. In order to execute the join point itself when using an `around` action, a special `proceed` call exists. This call can be used in the action code to execute the original join point. Omitting this call from action code results in the original join point never being executed.

ASPECTMATLAB allows for extraction of context-specific information about join points via the use of predefined context selectors. These selectors are specified along with an action definition, and allow for context-specific information to be used in action code. An example of context exposure is shown in Figure 2 on line 17, where we use the name selector, to expose the name of the function matched by the pattern. This information is then used on line 19 to display the name of the function being called. The applicable selectors depend upon the type of join point being matched.

3. Language Extensions

We have defined and implemented a variety of new patterns to match additional language constructs. Given that our target audience includes novice programmers, we wanted to include a pattern in ASPECTMATLAB that allows MATLAB programmers who may not be familiar with aspect oriented programming to exert more control as to where aspect code would be woven. To this end, we introduce the `annotate` pattern, which allows for pattern matching on special MATLAB comments. This pattern makes a useful tool for our own aspects, as it allows for general solutions to problems to be written as aspects, while making it possible the specifics to be written into the base code as part of annotations.

To address the difficulties that MATLAB's dynamic typing we introduced patterns for matching based on the size of arrays with the `dimension` pattern, as well as the type of data that arrays store, with the `type` pattern.

Another important extension was to allow for context exposure of the loop body in loop patterns through use of a special `body` call. This is essential for `around` advice on a loop pattern, which themselves are of the utmost importance when targeting scientific languages, as it makes it possible for us to alter the loop body while still executing the base code correctly.

3.1 Annotation Pattern

The annotation pattern differs from other patterns in ASPECTMATLAB in that it does not match on MATLAB code itself. Instead, we allow for programmers to write annotations which take the form of structured comments in their base code. The `annotate` pattern then matches these specially formatted comments. This makes it possible to provide information to the aspect program at any point in the execution, and allows for code to be woven easily into arbitrary points of a program, all without requiring any alteration to the execution of the base code. This new functionality makes it easy to write code with aspects in mind, by allowing for easy communication of relevant information from the program being cross-cut to the aspect.

Due to the fact these annotations will not be executed by a MATLAB runtime, this approach has the benefit of ensuring that the program will execute normally without having to weave aspects. The syntax for an annotation is shown in Figure 4. To specify that a particular comment should be recognized as an annotation, it is marked using the '`@`' symbol, and is followed by an identifier that gives the name of the annotation. Following the identifier is a list of arguments, whose values can be exposed as context in an action definition.

```
<annotation> ::=
      '%@' <annotation_name> <annotation_arguments>*
<annotation_arguments> ::=
      IDENTIFIER | STRING_LITERAL | CONSTANT
    | '[' <array_argument> ']'
<array_argument>::=
   <annotation_argument>
  | <array_argument> ',' <annotation_argument>
```

Figure 4. Syntax of MATLAB Annotation

There are four types of arguments that can be exposed as context, `var` (IDENTIFIER), `str` (STRING_LITERAL), `num` (CONSTANT), and arrays of other arguments. Exposure of a `var` provides the value of that variable as context to the aspect code. If a variable is undefined, it will instead be exposed as a value of class `AMundef`, an empty Matlab class. `str` exposes a string, and `num` a numeric value as a double. Arrays of arguments will expose a cell array containing the context exposed by those arguments.[1] All arguments adhere to standard Matlab syntax.

The syntax for the annotation pattern is shown in Figure 5. It takes the name of the annotation it should match, as well as a list of arguments that are expected to be present in the annotation. In the event that a pattern matches the identifier and arguments, but has arguments in excess of those specified, the pattern will still match, and the excess arguments will not be exposed as context. This allows a MATLAB programmer to easily integrate annotations into their comments, without having to omit information to fit the annotation formatting. The allowed arguments are those described above and an insufficient number of arguments will result in no match.

```
< annotation_pattern > ::=
  'annotate' '(' <annotation_name>
                 '(' < annotation_pattern_selector >* ')' ')'
< annotation_pattern_selector > ::=
      'var' | 'str' | 'num'
    | '[' < annotation_pattern_selector > ']'
    | '*'
```

Figure 5. Syntax of Annotation Pattern

An example of use of the annotation pattern is shown in Figure 6. In this case, the pattern will match all annotation comments that begin with the name 'type' and have at least three arguments: the first a variable name, the second a string, and the third an array of numbers.

3.2 Type Pattern

The `type` pattern, introduces matching on the base type of arrays to ASPECTMATLAB. For this pattern, a base type can be one of

[1] Cell arrays in MATLAB allow for different elements in the array to have different types, and so can be used to represent heterogeneous collections.

```
patterns
  typepat : annotate(type(var, char, [num]);
end
```

Figure 6. Example of Annotation Pattern

several MATLAB defaults, such as `double`, `char`, `int32`, or it can be a user defined class type. The `type` pattern captures all variable accesses and assignments in which the variable matches a specified MATLAB base type. The syntax for invoking the type pattern is `type(<basetype>)`, where `basetype` is the name of the MATLAB base type to be matched. Accesses and assignments are captured in the order of evaluation of an expression, with subexpressions being matched before their containing expression. For assignments, the MATLAB type considered is the one which would be held after assignment occurs.

To match accesses or assignments of specific arrays, but only when they are of a particular type, one can use a compound pattern of the `type` and `get` or `set` patterns. Examples of `type` patterns are given in Figure 7. Pattern `isint32pat` matches all join points where an array access or assignment is being performed involving an array of type `int32`. Pattern `isxsinglepat` demonstrates a compound pattern using `type`, `get`, and `set` to match all accesses and assignments to array `x`, when `x` is of type `single`.

```
patterns
  isint32pat : type(int32);
  isxsinglepat : (get(x)|set(x))&type(single);
end
```

Figure 7. Example of Class Pattern

In addition to the standard MATLAB base types, we have defined one further base type called `realint` to capture the special issue in MATLAB with its use of positive real integers. As the default data type in MATLAB is a double, all arrays can be indexed using double values. However, in the event that this double value does not correspond to a positive integer, this will return an error. To determine if double `x` is a real integer or not, it is necessary to check that the `x = round(x)`. The inclusion of this base type makes it easy to determine whether the value in a matched array can be used as an index.

3.3 Dimension Pattern

Due to the fact that arrays are the default data type in MATLAB, it can be helpful to identify arrays by their size. To this end we introduce the `dimension` pattern. The `dimension` pattern is similar to the `type` pattern, but instead of matching join points based on type, it instead matches by the dimensions of the associated vector.

The `dimension` pattern takes as arguments the size of the dimensions of the matrix it should match. Similarly to the `type` pattern, the `dimension` pattern matches variable accesses and assignments when the array is of the shape specified by the pattern's arguments. The syntax for invoking the dimension pattern is `dimension(<dimensions>)`, where `dimensions` is a comma separated list of the expected dimensions of the array. For each dimension, an integer value corresponding to the expected size of the dimension may be specified, or the wild-card symbol, '`*`' may be used to indicate that a dimension can be any size. As with the `class` pattern, accesses of specific arrays can be accomplished by using a compound pattern of the `dimension`, `get` and `set` patterns.

Examples of the `dimension` pattern are shown in Figure 8. Pattern `dimp` will match all array accesses and assignments involving arrays that have 3 dimensions, and whose first dimensions is of size 2. Pattern `dimx2by2` will match array accesses or assignments to `x` when `x` is a 2 by 2 matrix.

```
patterns
  dimp : dimension(2,*,*) ;
  dimx2by2 : (get(x)|set(x))&dimension(2,2);
end
```

Figure 8. Example of Dimension Pattern

3.4 Loop Body Context Exposure

Loops are a critical structure to target when writing an aspect language targeting scientific programming. To ensure these constructs could be handled meaningfully, the original ASPECTMATLAB introduced several patterns which matched on loops. While it did successfully introduce means of handling loops themselves, one significant shortcoming is that it did not introduce any means of handling or restructuring the bodies of loops. To allow ASPECTMATLAB programmers to deal with the body of loops more precisely, we introduce the `body` call.

ASPECTMATLAB features a special `proceed` call which can be used in `around` advice to execute the original join point. While useful in most scenarios, with loop patterns it would be useful to be able to execute simply the body of the loop, as opposed to the entire join point. The `body` call is similar to the `proceed` call, however, instead of executing the entire join point, it simply executes a portion of it - that portion which corresponds to a single execution of the body of a loop. To interact with the contents of the body, we also introduce the `loopiterator` keyword, which can be assigned a value to replace or modify the value held by the loop iterator variable.

The example in Figure 9 showcases the use of these keywords, with an aspect that replaces loops which iterate over `i` with a loop which iterates over the square root of the base MATLAB loop's arguments. The `args` exposed as context are the loop iteration space, which is then square rooted and passed in as a loop iterator for another for loop, which makes a call to `body` to execute the body of the original loop.

```
actions
  sqrtiter : around loop(i) : (args)
    for loopiterator = sqrt(args)
      body()
    end
end
```

Figure 9. Example of Loop Body Context Exposure

4. Scientific Aspects

Using the language extensions outlined in the previous section, we designed and implemented a number of scientific aspects to help programmers to better understand and work with their MATLAB code. In order to ensure programs meet expected type requirements, we introduce a *type checking aspect*, which allows for programmers to specify stricter types using type annotations which are matched by the `annotate` pattern. The *unit checking aspect* uses unit annotations to allow programmers to ensure that their units match, following basic rules of dimensional analysis. This allows for type checking of a scientific variety. In order to better deal with peculiarities of MATLAB, we also introduce two *type profiling aspects*,

which employ the `class` and `dimension` patterns to detect dynamic type information. To help programmers get the most out of MATLAB code, we also use aspects to automate loop transformations which use the newly introduced body context exposure for loops, improving the ability of ASPECTMATLAB to help programmers increase the efficiency programs by making *loop unrolling* and *loop reversal* optimizations quickly and easily.

4.1 Stricter Type Checking

In MATLAB, variable types are not declared, and are instead determined dynamically. This is beneficial for fast prototyping, as it allows programmers to focus on the task at hand. Despite MATLAB's lack of static type checking, programmers will often have some assumptions about the types being used in their programs, as shown previously in Figure 1. Using mechanisms provided by MATLAB in order to ensure that the types of variables are correct after every assignment would require the programmer to insert checks manually, a tedious job which opens up room for errors. The significant number of superfluous checks would negatively impact the performance.

Our solution to this problem is to leverage the power of the newly introduced `annotate` pattern to allow programmers to format their comments in such a way that the type information they contain can be confirmed by an aspect. Figure 10 shows an example of what annotated code looks like. It is very similar to the original comments, but by formatting the comments into type annotations, the information in the comments can be used by the ASPECTMATLAB compiler.

```
1  function [F, V] = nbody3d(n, R, m, dT, T)
2  %-------------------------------
3  % This function M-file simulates the gravitational
4  %   movement of a set of objects
5  % Invocation:
6  %   >> [F, V] = nbody3d(n, R, m, dT, T)
7  %   where
8  %inputs:
9  %@type n 'double' [1,1] %number of objects
10 %@type R 'double' [n,3] %matrix of radius vectors
11 %@type m 'double' [n,1] %vector of object masses
12 %@type dT 'double' [1,1] %time increment of evolution
13 %@type T 'double' [1,1] %total time for evolution
14 %outputs:
15 %@type F 'double' [n,3]  %matrix of net forces,
16 %@type V 'double' [n,3]  %matrix of velocities.
17 %-------------------------------
```

Figure 10. Header of MATLAB simulation of n-body problem with type annotations

The formatting of these annotations are specified by the `annotate` pattern in our type checking aspect and operates as shown in Figure 11. As with all ASPECTMATLAB annotations, the first piece of information is annotation name, "type" which is followed by three other arguments. These are the identifier for the variable it should be checking, a string which declares the base type the variable is expected to have, and an array that lists the expected size of each dimension.

As an example, the annotation `%@type n 'double' [1,1]` is checking that the variable n is of type double, and is a scalar (or 1 by 1 matrix). For the base type, the user may require that the variable be of any base MATLAB datatype, as well as any user defined datatypes. The dimensions can be expressed in one of three ways, which capture the variety of possible requirements a programmer may have of their program. The simplest method of communicating the dimensions is by simply specifying the expected size numerically in the annotation. In this case, the aspect will ensure that the

```
⟨type annotation⟩ ::⇒
    '%@' 'type' ⟨variable⟩ '⟨type⟩' '[' ⟨dimensions⟩']' ⟨comment⟩*
⟨type⟩ ::⇒
    ⟨base matlab datatype⟩
    | ⟨user defined datatype⟩
⟨dimensions⟩ ::⇒
    ⟨dimension⟩ ',' ⟨dimensions⟩
    | ⟨dimension⟩
⟨dimension⟩ ::⇒
    IDENTIFIER | DOUBLE | CHAR
```

Figure 11. Definition of a type annotation

size of the variable's dimension matches the one specified by the programmer. This is demonstrated on line 1 of Figure 12. The second method for specifying dimension size is by using an identifier which corresponds to a variable in the base MATLAB code, which contains the size it should match. This is shown on line 2 of Figure 12, where variable `b` is required to be a matrix of size n by m, where n and m will have been previously initialized in the base MATLAB code. Alternatively, a string can be used to specify the dimension sizes, as shown on lines 3,4 and 5 of Figure 12. When a string is used, the aspect checks that the size of the dimension matches any other dimensions which use the same string. For example, variable `c` is required to have 2 dimensions, and because both are specified by the same string, both dimensions must have the same size. Similarly, the annotations on lines 3 and 4 specify that the first dimension of `d` must be the same size as the second dimension of `e`, and vice versa.

```
1 %@type a 'double' [2,3,4]
2 %@type b 'userdef' [n,m]
3 %@type c 'char' ['x ',' x']
4 %@type d 'double' ['y ',' z']
5 %@type e 'double' ['z ',' y']
```

Figure 12. Example of type annotations

Once the aspect has been woven, the aspect code will check whether or not the types hold at the program point where the annotation is present. In addition, any time a variable is assigned to after the annotation, the aspect code will check again to ensure that it obeys the specified restrictions. Not only are these annotations simple and straightforward, able to be easily inserted anywhere in a program where a programmer is uncertain, but they correspond closely to comments that are typically found in MATLAB programs.

The type checking aspect, outlined in Figure 13, takes advantage of the annotation pattern and weaves type checking code around these annotations. There are only two patterns required for this aspect. The pattern `typeAnn` matches the type annotations. It matches annotations with the annotation name type, and takes three arguments, an identifier corresponding to the variable in question, a string constant corresponding to its type, and an array which can contain any combination of numerical constants, identifiers, or string constants, corresponding to its dimensions as described above. This demonstrates that while the annotation pattern is simple to use, it can be very powerful, allowing for easy introduction of language features that don't otherwise exist in MATLAB.

From here, the code which carries out type checking is straightforward. The action `actAnn`, given in Figure 14, uses the information provided by the annotation. This information is extracted using the `args` context selector, which yields a cell array of the arguments' values, as well as the `rep` context selector, which provides a cell array of string representations of the arguments' values. Before we type check, we confirm that the variable in question has been assigned a value. If it does not, the content of the variable will be of type `AMundef`, signifying that no definition has

```
aspect typechecking
  ...

  patterns
    typeAnn : annotate(type(var, char, [*]));
    arraySet : set(*);
  end

  actions
    actAnn : before typeAnn : (args, rep, line)
      value = args{1};
      varname = rep{1};
      expectedclass = args{2};
      expecteddims = args{3};
      %Check that the provided variable meets the
      %specified type and dimensions
      %Store the type and dimensions for future checks
        ...

    actArraySet : after arraySet : (newval, name, line)
      %Compare with known type and dimensions for the
      %given variable name
       ...
  end
end
```

Figure 13. Outline of type checking aspect

```
1  actAnn : before typeAnn : (args, rep, line);
2    %context expose of args gives the arguments provided by the
3    %annotation in order
4    %context expose of rep gives the string representation of the
5    %annotation arguments, which we use to obtain a string
6    %representation of the variable
7    value = args{1};
8    varname = rep{1};
9    expectedclass = args{2};
10   expecteddims = args{3};
11
12   %if the variable is defined at the point of the annotation, we
13   %immediately perform type checking
14   if (¬isa(value, 'AMundef'))
15     dimensions = dims(value);
16     if ((ndims(value) ≠ size(expecteddims,1)) )
17       %prepare error message
18     end
19     if (¬isa(value, expectedclass))
20       %prepare error message
21     end
22
23     for dim = 1:size(expecteddims,1)
24       if (isa(expecteddims{dim},'char')
25         %check against any previous definition for the string
26         if (ismember(expecteddims{dim}, keys(chardims)))
27           if (size(value,dim)≠ chardims(expecteddims{dim}))
28             %prepare error message
29           else
30             %associate value with string if no previous definition
31             %exists
32             chardims(expecteddims{dim}) = size(value,dim);
33           end
34         elseif (size(value,dim) ≠ dimensions{dim})
35           %prepare error message
36         end
37       end
38       %emit error message if one exists
39     end
40     %Whether the variable has been defined, we associate the
41     %variable name with the expected class and dimensions
42     varclass(varname) = expectedclass;
43     vardims(varname) = expecteddims;
44   end
```

Figure 14. actAnn action of typechecking aspect in detail

been provided and that we can proceed to storing type information. If the variable in question has been defined, we check that its value has the expected number of dimensions, the correct type, and the correct size for each dimension as specified by the annotation. In the event that there is a conflict, an error is emitted. If a string has been used to specify the size of a dimension, we check to see if any previous annotations used the same string - comparing size with the previous use if it has, and associating the current value for our variable if it hasn't. Regardless of whether type checking is performed, the information from the annotation is stored in `container.Map` objects, so future array assignments can be checked. The action `actArraySet` , which is woven around assignments to arrays, checks whether or not the variable being assigned to has an existing mapping from a previous type annotation. If it does, type checking is performed using similar checks to those made in the `actAnn` pattern, and throwing an error if the checks fail.

Using this aspect, it is possible to leverage the types specified by the programmer, and throw meaningful errors when they are not met. Due to the fact that types are stored as annotations, it is possible for a programmer to simply run their program without weaving type checking code once they are certain that their program will execute correctly. This allows for programmers to enable type checking at a small cost in performance to ensure their MATLAB code executes as expected, and later dispense with the type checking to ensure optimal performance. Given that these annotations require no significant knowledge of ASPECTMATLAB to introduce into a program, this aspect provides incentive for those who are not interested in learning the ASPECTMATLAB language to use its functionality.

4.2 Units of Measurement as Types

In the previous section we explored an aspect for handling traditional type checking, however, for scientific programmers there exists more type information presented in Figure 1 aside from the dimensions of variables. Many inputs may have associated units of measurement, corresponding to physical qualities such as times, distances, and masses. Even in a programming language such as MATLAB, which targets scientific programmers, these units of measurement will be lost, as it is most easy and efficient to store data as doubles instead of having a separate class for every piece of data. This is unfortunate, as these units of measurement have meaning within the context of a program, particularly programs with a large number of arithmetic operations. For example, we know that it makes sense to add a distance to a distance, and the result of such a computation would be a distance itself. However, it is not meaningful to add a distance to a mass, as the result of the computation would be meaningless. Similarly, assigning a value that is known to be a mass to a variable which should contain a distance would also be incorrect. By keeping track of the units that variables are intended to have, it is possible to prevent programmers from making mistakes by throwing errors when incorrect units are used. In this sense, units can be thought of as types unto themselves. By providing programmers with a means to incorporate units into their programs, it can be ensured that their programs only use them safely.

In order to take advantage of unit information, we introduce the the idea of unit annotations. Figure 15 shows how unit annotations can be used to take advantage of the information programmers provide. Similarly to the type aspect, our unit aspect defines a formatting which unit annotations must follow, and this is shown in Figure 16. Following the annotation name, two arguments are

required. The first is the string corresponding to the variable whose units we are keeping track of. The second is an array of strings containing the SI units of the variable. For example, the annotation `%@unit 'dT' ['s']` specifies that the variable dT is a measure of time, in seconds. We support several common derived units, and for those unsupported, combinations of the 7 base SI units can be used. For example, a newton which could be conveyed in an annotation as `['N']`, may also be expressed as $kg \cdot m/s^2$ which could be written in an annotation as `['kg','m','s^-2']`. Exponents can be used in unit annotations, and units may be provided in any order.

```
1  function [F, V] = nbody3d(n, R, m, dT, T)
2  %------------------------------
3  % This function M-file simulates the gravitational
4  %   movement of a set of objects
5  % Invocation:
6  %   >> [F, V] = nbody3d(n, R, m, dT, T)
7  %   where
8  %inputs:
9  %@unit 'n' [] %number of objects
10 %@unit 'R' ['m'] %n x 3 matrix of radius vectors
11 %@unit 'm' ['kg'] %n x 1 vector of object masses
12 %@unit 'dT' ['s'] %time increment of evolution
13 %@unit 'T' ['s'] %total time for evolution
14 %outputs:
15 %@unit 'F' ['N']
16 %@unit 'V' ['m','s^-1']
17 %------------------------------
```

Figure 15. Header of MATLAB simulation of n-body problem with unit annotations

```
⟨unit annotation⟩ ::⇒
    '%@' 'unit' '⟨variable⟩' '[' ⟨units⟩ ']' ⟨comment⟩*
⟨units⟩ ::⇒
    ⟨unit⟩ ',' ⟨units⟩
    | ⟨unit⟩
⟨unit⟩ ::⇒
    | '< si_unit >'
    | '< si_unit > '^' INTEGER '
⟨<si_unit>⟩ ::⇒
    m | kg | s | A | K | mol | cd | N | J | ...
```

Figure 16. Definition of a unit annotation

The unit checking aspect works similarly to the type checking aspect, taking advantage of these annotations the programmer can use in their code. We have a usage of the `annotate` pattern which matches all unit annotations, as they have been specified. Around these annotations, we weave code which stores the type information and ensures that there is no conflict with the current units held in the variable and the ones the annotation specified. We apply these units as specified by the annotation when they are accessed by using the `get` pattern, and store these units in a struct which contains both the units as well as the original value. The original value is stored in a val field, and the units are stored in a unit field as an array containing the exponents of each of the 7 base SI units. For example, a value of 3 newtons would be converted to base SI units, $kg \cdot m/s^2$, and stored as `struct('val',3,'units',[1,1,-2,0,0,0,0])`. Wrapping all units in such a structure would result in disruption of the normal program execution. To handle this, we perform a unit removal whenever we cannot be certain that such a struct would be handled appropriately, for example when it is used as a parameter in function calls, or used as an index for an array. We also match all arithmetic operations, and, using separate actions for each operation, determine what the units will be after a line of code is executed. In the case of addition, subtraction and exponentiation, we also ensure that there is no conflict in the units of the operands. From the point of a unit annotation onward, any addition or subtraction operations between a specified variable and another variable that has not been annotated with the same units will result in an error, and the programmer will be informed of the mismatch. Similarly, exponentiation performed with an exponent that has units will also result in an error. Using the `set` pattern, we check on each assignment whether a unit annotation has been violated, throwing an error when an assignment is made to that variable that has units other than those specified.

As with the type checking aspect, the use of annotations allows a programmer to run the program with the unit check on, ensure no unit violations take place, and then run the program without the aspect afterwards to avoid the overhead associated with unit checking.

4.3 Profiling Real Positive Integers

The default numeric data type in all MATLAB programs is a double-precision floating point. This has some advantages for scientific programmers, as doubles are more likely to be used in most scientific computations. Not having to worry about conversions between reals and integers allows scientific programmers to focus on the functionality of their program. One negative side effect of this, however, is that all arrays in MATLAB can be indexed using real numbers. Given that indexing with a non-integer value is not meaningful, MATLAB only allows certain classes of real numbers to be used for indexing, "real positive integers".

A *real positive integer* is a floating point number which corresponds to an integer value. Checking whether or not a particular variable is a *real positive integer* at all program points is a tedious task for a programmer. However, making a mistake in using a value that is not a real positive integer when one is required will result in unexpected runtime errors. In addition, it presents some difficulty for those interested in translating MATLAB programs to other languages, as it cannot necessarily be checked statically. To aid these two groups, an aspect was created to determine which variables cannot be safely used for indexing.

This aspect, outlined in Figure 17 takes advantage of the type pattern, matching on all array accesses which do not contain real integers. By taking note of the variables which are not real integers, we can know which variables would be unsafe to use for array access operations. The pattern `realint` matches all array accesses which are not real positive integers. The action `actRealInt` uses the `name` and `line` context selectors to store the name of the variable being accessed, as well as the line number at which it was accessed. At the end of the execution of the program, the `actRealInt` action prints the result of the profiling, and informs the programmer as to which variables were unsafe, and the lines where they were found to be unsafe.

4.4 Profiling 1 Dimensional Arrays

Another area where profiling MATLAB is useful is array dimensions. As MATLAB is a matrix-based programming language, a one dimensional array can refer to either a matrix with only 1 column and several rows or a matrix with only 1 row and several columns. This differs from other more conventional programming languages, where information stored in a one dimensional array does not have an orientation. While the data contained in a one dimensional array may seem the same to a programmer regardless of orientation, the orientation can change the functionality of a program. An improperly oriented array could lead to program terminating runtime errors, or to data being improperly handled, causing incorrect results.

To aid programmers in understanding their code, we provide an aspect which profiles one dimensional arrays was written, as outlined in Figure 18. We match on all assignments of one dimensional

```
aspect safeindex
  properties
   nonrealint = container.Map;
  end

  patterns
     realint : get(*)&class(¬ realinteger );
    exec : mainexecution();
  end

  actions
    actRealInt : after realint : (name,line)
      %Store the name and line of the variable, if it
      %has not already been stored.
       unrealintlines = nonrealint[name]
      if(¬ismember( unrealintlines , line ))
        nonrealint[name] = [ unrealintlines line ]
      end

    results : after exec
     %Display results
      ...
  end
end
```

Figure 17. Outline of real integer profiling aspect

arrays, and store information about the orientation and size of the array. At the end of execution, of the program, the programmer is informed of the orientation of the arrays. Weaving and running this aspect provides the programmer with certainty that their arrays are handled correctly.

```
1  aspect profileoned
2    properties
3      columnvars = container.Map;
4      rowvars = container.Map;
5    end
6
7    patterns
8      onedcolumn : set(*)&dimension(1,*);
9      onedrow : set(*)&dimension(*,1);
10     exec : mainexecution();
11   end
12
13   actions
14     actColumn : after onedcolumn : (name,line)
15       %Store the name of the variable, the line it
16       %occured on, and its orientation
17       columnslines = columnvars[name]
18       if(¬ismember(columnslines,line))
19         columnvars[name] = [columnslines line]
20       end
21
22     actRow : after onedrow : (name,line)
23       %Store the name of the variable, the line it
24       %occured on, and its orientation
25       rowslines = rowsvars[name]
26       if(¬ismember(rowslines,line))
27         rowvars[name] = [rowslines line]
28       end
29
30     results : after exec
31       %Display results
32
33   end
34 end
```

Figure 18. Outline of one dimensional array profiling aspect

The aspect itself consists of two primary patterns, both of which make use of the `dimension` pattern. `onedcolumn` catches all array assignments which correspond to a 1-dimensional column, and `onedrow` does the same for 1-dimensional rows. The actions `actColumn` and `actRow`, are performed after the assignments to the 1-dimensional array are made. The name and line context selectors are used so we can store information about the variable being assigned to, and the location in the code where the assignment took place. The action `results`, called after the execution of the program, prints out a list of assignments made to 1-dimensional arrays over the course of the program, consisting of the name, the line it occurred on, and the orientation.

4.5 Loop Transformations with Aspects

In addition to the aspects which help manage type information, we have also contributed new aspects which the goal of helping programmers improve the efficiency of their code. One such aspect is the loop unrolling aspect. Loop unrolling is a fairly simple transformation, in which the body of a loop is executed several times per loop iteration. The result is that the loop condition needs to be tested fewer times, and fewer jumps are required. The cost of this transformation is that the program's code size increases, and may experience slowdown due to poor register usage. However, unrolling loops by hand is tedious and time consuming, requiring large amounts of code to be copied and repeated. In addition, if loop unrolling decreases performance, additional effort must be expended to repair the code to its original state. This additional programming overhead decreases the appeal of this transformation.

In order to automate the process, and decrease the effort required to determine whether manual loop unrolling is a beneficial transformation, we introduce an aspect which unrolls for loops. This aspect takes advantage of newly introduced loop functionality, as well as annotations. In order to unroll the desired number of times, the unroll aspect takes information from an unroll annotation, which the programmer may place in their code, and which takes the form `@unroll 10`. It takes as an argument the number of times the loop should be unrolled, and from the point of the annotation onward, for loops will be unrolled the specified number of times. Due to limitations of the ASPECTMATLAB engine, the only accepted values for the unrolling parameter are 2,3,4,5, and 10. The `body()` call is used to duplicate the loop body multiple times. The result of weaving the aspect is that each loop will be unrolled to 5 extents (2,3,4,5, and 10), and at runtime, the loop which matches the desired number of unrolls will be run.

```
1  aspect reversal
2
3    actions
4      reversal : around loopp : (looptype, obj)
5        if(strcmp(looptype,' for '))
6          i = size(obj);
7          while(i>0)
8            loopiterator = obj(i)
9            body()
10           i = i-1;
11         end
12       end
13   end
14 end
```

Figure 19. Outline of loop reversal aspect

Another loop transformation that can be performed with aspects is loop reversal. The transformation involves reversing the order in which values are assigned to the index variable. For example, instead of iterating over the set of values from 1 to 100, we iterate from 100 to 1. Like loop unrolling, loop reversal is a fairly simple

optimization, but performing it by hand on each loop to determine its value is time consuming. Again, we introduce an aspect which automates the process of reversing loops. This aspect functions by executing the body with a loop iterator that proceeds in reverse. This aspect, given in Figure 19 and which involves only a few lines of aspect code, demonstrates the ease with which transformations can be executed.

5. Implementation

While the original implementation of ASPECTMATLAB provided a clean means for weaving aspect-oriented code into MATLAB, the implementation, making use of MATLAB classes, was particularly slow. Aspect code was converted into MATLAB classes, with action code being converted into MATLAB class methods. Properties, as well as methods used by aspect code remain part of the MATLAB class after compilation. These properties and methods are then referenced from the appropriate join points in the base code. Unfortunately, object-oriented MATLAB classes require a significant amount of overhead to access their methods and properties, enough to cause woven code to run many times slower than the base code on its own. In order to eliminate this slowdown of aspect code, we inline the action code, and make local copies of aspect properties to avoid unnecessary overhead.

To demonstrate our enhancements, we look at the weaving of a simple aspect which counts flops, shown in Figure 20, and apply it to a heat equation solver in Figure 21, the result of which is shown in Figure 22.

```
aspect flopcount
    properties
        count = 0;
    end

    patterns
        flopp : op(*);
    end

    actions
        aflop: before flopp
            count = count + 1;
        end
    end
end
```

Figure 20. Simple aspect to count all floating point operations

```
function solveHeatEquation(a, steps)
  tN= 3; % end of time interval
  N = 300; % set total steps
  h = 2*pi/(N-1); % set spacial step
  X = [h:h:2*pi]; % set X axis points..
  U0 = 0*X; % set initial condition
  U0(round(end/2.2):round(end/1.8)) = 1;
  D = (Dxx(N)/h^2); % set second spatial derivative matrix
  function y = F(t,u) % set rhs of ODE, i.e. Ut
    y = a*D*u;
  end
  W = RungeKutta4(@F,[0, tN],U0,steps,1); % find the solution
  disp('computation finished');
end
```

Figure 21. Matlab function which solves the heat equation

After weaving, the methods corresponding to action code would be called from the join points which match the associated patterns.

```
classdef flopcount < handle
  properties
    count = 0;
  end
  methods
    function [] = flopcount_aflop(this)
      count = (count + 1);
    end
  end
end
```

Figure 22. Generated code for the flopcount aspect for Figure 20

5.1 Inlining of Action Code

While the weaving shown in Figure 22 provides a functional solution, calling a method from a MATLAB class entails a significant amount more overhead than a typical MATLAB function call. Looking at the woven code, shown in Figure 23, it can be noted that there are many calls to the method `flopcount.flopcount_aflop()`.

```
function [] = solveHeatEquation(a, steps)
  global AM_GLOBAL;
  if isempty(AM_GLOBAL)
    AM_GLOBAL.flopcount = flopcount;
    AM_EntryPoint_0 = 1;
  else
    AM_EntryPoint_0 = 0;
  end
  tN = 3; % end of time interval
  N = 300; % set total steps
  AM_tmpBE_0 = 2
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_1 = (AM_tmpBE_0 * pi)
  AM_GLOBAL.flopcount.flopcount_aflop();
  h = (AM_tmpBE_1 / (N - 1)); % set spacial step
  AM_tmpBE_2 = 2
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_3 = (AM_tmpBE_2 * pi)
  X = ([(h : h : (AM_tmpBE_2 * pi))]); % set X axis points..
  AM_tmpBE_4 = 0
  AM_GLOBAL.flopcount.flopcount_aflop();
  AM_tmpBE_5 = (AM_tmpBE_4 * X)
  U0 = AM_tmpBE_5; % set initial condition
  AM_GLOBAL.flopcount.flopcount_aflop();
  U0((round((end / 2.2)) : round((end / 1.8)))) = 1;
  AM_GLOBAL.flopcount.flopcount_aflop();
  D = (Dxx(N) / (h ^ 2)); % set second spatial derivative matrix
  W = RungeKutta4(@F, [0, tN], U0, steps, 1);
% find the solution
  disp('computation finished');
  if AM_EntryPoint_0
    AM_GLOBAL = [];
  end
end
```

Figure 23. Example of code woven before inlining of action code

To determine the impact of these calls to MATLAB classes, we test the overhead of the call itself by making calls to empty functions. In Figure 5.1, we show the time required for a call to an ordinary MATLAB function `empty()`. This is compared with calls to a class method, `empty(obj)`, which can also be invoked by calling `obj.empty()`. We also compare to calls to a static class method `test.empty()`. These results demonstrate that calls to object-oriented MATLAB code are more than 12 times slower than ordinary function calls, and that static methods are slower still.

To increase the performance of woven aspect code, it would be ideal to do away with these method calls. One possible solution would be to performing a simple inlining on the function calls

Table 1. Function call overhead for methods in object-oriented MATLAB

	Time (s) for 1000000 calls	Time (μs) per call
empty()	0.104	1.04
empty(obj)	1.294	12.94
obj.empty()	1.922	19.22
test.empty()	2.473	24.73

woven into the base code. This would require further work to be done in order to inline around actions, where we have to deal with proceed calls that require passing of context information. A simpler option is to weave the action code directly into the base code without ever writing it into the aspect class as a method.

We do this by performing a renaming on the action code, to ensure that operations local to the aspect do not interfere variables in the base code, and then insert the action code can be inserted directly into the AST. Assignments to set up context selectors used by the action code are inserted before the action code itself in the AST.

5.2 Considerations for Around Advice

Special consideration must be taken for the case of around advice. Due to the fact that the original join point is replaced when using around advice, the original join point must either be removed from the AST, or placed at the location of a proceed call if one is made. In addition, we also have to consider the fact that around actions can return data, and the data returned should correspond to the result of the execution of the original join point. To accomplish these tasks, we replace proceed calls with the original join point, and set the result aside, to be returned as a result at the end of the around advice.

The `body` calls are handled similarly to `proceed` calls, with the difference being that we replace the call to `body` with the entirety of the body of the loop.

It is worth noting that the previous implementation's handling of around advice required passing of context information to action code. In order to ensure that the join point is correctly executed when a proceed call is made, it was necessary to store all necessary context information within the action method, and use a switch statement to refer to the context of specific join points. By writing the action code directly into the AST of the base code, instead of making a call to a class method, this overhead is unnecessary. We can simply execute the join point with its original context intact, resulting in further performance enhancements when several context selectors are used.

5.3 Local Copies of Aspect Properties

With action code having been inlined, aspect properties referenced in action code must still be accessed through the generated MATLAB class. An example of this can be seen in 24, where calls to a counter stored in the MATLAB class are performed before every flop. While the inlining process does reduce a significant amount of the overhead involved in woven code, the overhead from accessing these object properties may still be significant. We carried out a study to determine the impact of property accesses and assignments by performing each 1000000 times within a loop. The results are shown in Figure 5.3. While not as significant as the overhead for calls to object-oriented methods, property accesses and assignments still take up more time than function calls to ordinary MATLAB code, and are much slower than assignments to local variables.

In order to further reduce the overhead due to frequent accessing of object properties it is helpful to make local copies of properties

```
1  function [] = solveHeatEquation(a, steps)
2    global AM_GLOBAL;
3    if isempty(AM_GLOBAL)
4      AM_GLOBAL.flopcount = flopcount;
5      AM_EntryPoint_0 = 1;
6    else
7      AM_EntryPoint_0 = 0;
8    end
9    tN = 3;  % end of time interval
10   N = 300;  % set total steps
11   AM_tmpBE_0 = 2
12   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
13   AM_tmpBE_1 = (AM_tmpBE_0 * pi)
14    AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
15   h = (AM_tmpBE_1 / (N - 1));  % set spacial step
16   AM_tmpBE_2 = 2
17   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
18   AM_tmpBE_3 = (AM_tmpBE_2 * pi)
19   X = ([(h : h : (AM_tmpBE_2 * pi))]);
   % set X axis points - the first point is ommitted (0)
20   AM_tmpBE_4 = 0
21   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
22   AM_tmpBE_5 = (AM_tmpBE_4 * X)
23   U0 = AM_tmpBE_5; % set initial condition
24   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
25   U0((round((end / 2.2)) : round((end / 1.8)))) = 1;
26   AM_GLOBAL.flopcount.count = AM_GLOBAL.flopcount.count + 1;
27   D = (Dxx(N) / (h ^ 2));  % set second spatial derivative matrix
28   W = RungeKutta4(@F, [0, tN], U0, steps, 1);
   % find the solution
29   disp('computation finished');
30   if AM_EntryPoint_0
31     AM_GLOBAL = [];
32   end
33 end
```

Figure 24. Example of code woven after inlining of action code

Table 2. Property access and assignment overhead for object-oriented MATLAB

	Time (s) for 1000000 calls	Time (μs) per call
property access	0.366	3.66
property assign	0.286	2.86

whenever possible. By doing this, base code has many matched join points, or action code which makes reference to the same property several times, can be rendered more efficient. One precaution we must take when making local copies is ensuring that the properties are updated before they are used or modified elsewhere in the code. To ensure this, we must copy back to the object prior to any function calls to code that is woven by the same aspect. Similarly, we must be certain to copy the property back to the associated object before the end of the woven function as well.

6. Performance

We provide the results of experiments performed on a set of MATLAB benchmarks, which demonstrate the significance of the optimization made to the ASPECTMATLAB compiler, as well as the utility of the scientific aspects provided. The benchmarks used are real-world programs, which have come from various projects targeting MATLAB, such as FALCON [14] and OTTER [12] , Chalmers University of Technology[2] and "The MathWorks' Cen-

[2] http://www.elmagn.chalmers.se/courses/CEM/

Table 3. Time in seconds for execution of benchmarks, and slowdowns with woven aspects

Benchmark	Time (sec)	Slowdown					
	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling	Loop Unrolling	Loop Reversal
beul	24.95	1.65	3.78	1.42	1.29	-	-
crni	23.56	1.29	7.23	1.34	1.24	0.97	1.05
diff	23.82	3.20	3.75	1.39	1.28	1.18	1.11
fdtd	25.23	3.58	16.15	1.54	1.48	0.96	-
fiff	24.50	1.30	1.24	1.36	1.34	0.98	0.97
hnor	26.12	1.04	-	1.18	1.19	1.00	1.01
mils	25.05	1.17	-	1.55	1.52	-	-
nb1d	25.19	3.69	5.88	1.53	1.50	1.04	-
nb3d	24.67	3.53	6.60	1.53	1.42	1.08	-
capr	24.78	1.29	8.10	1.79	1.71	0.98	-
Geometric Mean		2.00	5.43	1.46	1.39	1.03	1.03

tral File Exchange"[3]. To evaluate performance, we run each of the original benchmarks in MATLAB without weaving any aspects. We then weave code for each of the scientific aspects we have developed using the ASPECTMATLAB compiler, and run the woven code in MATLAB. Our performance analysis compares the running time of several benchmarks with each aspect. We then look at the impact of the optimizations outlined in Section 6 by performing the same experiment with them disabled. For our analysis, we ran all programs on MATLAB version 2013a, and used an Intel Core i7-3820 CPU @ 3.60GHz x 8 processor and 16 GB memory machine running Ubuntu 12.04 LTS.

6.1 Experimental Results

In Table 3, we list the execution time and slowdown of several benchmarks[4].

Given that most of our aspects involve several additional operations per operation in the base code, it is expected that we see some significant reducing in performance after weaving. We can note that both the type checking and unit checking aspects result in notable increases in runtime, with the type checking aspect running 2 times slower than the original program, and the unit checking aspect running 5.43 times slower than the original program. These slowdowns are the result of a large number of checks that must be made at every assignment to ensure the program is behaving as specified. In particular the fdtd benchmark runs 16.15 times slower when woven with units. The reason behind this is that there are many quantities which can be annotated with units, and as a result most computations require several secondary computations to ensure unit consistency. This may seem cumbersome, but one must consider that once a user has ensured their program is operating as expected it is simple to cease usage of these aspects. The profiling aspects also feature a notable slowdown, running at 1.46 and 1.39 times slower than original program. However, these slowdowns are reasonable considering their application.

Unlike the other aspects, we hope that the loop unrolling and loop reversal aspects may lead to some performance increase, as their purpose is to perform some optimizations prior to running the program. While on average neither of these aspects outperforms the base MATLAB code, we can note that the Loop Unrolling aspect runs faster for three benchmarks, crni, fdtd, and fiff, and that the Loop Reversal aspect runs faster on fiff. While the aspects result in a slowdown on average, it is easy to weave a program with these aspects, and if it performs better, continue using woven code or use it as an indication that one could consider performing the optimization manually. Otherwise, the aspects can simply be abandoned.

6.2 Analysis of Engine Improvements

In this section, we demonstrate the value of the engine improvements made to the ASPECTMATLAB compiler. First, we perform the same experiments as in the previous section, but we disable the function inlining and the local copy of aspect variable optimizations outlined in section Section 5. Of note, we do not perform this comparison with the loop optimization aspects, as the `body` keyword requires that these optimizations be in place in order to function. The results of this experiment are shown in Table 4. From these results, it's clear that the two optimizations have a significant impact. The unit checking aspect runs at 74.05 times slower than the base MATLAB program, making its use impractical. The type checking aspect, while not as bad, still performs 43.23 times slower, which again is an unreasonably large slowdown that makes it too cumbersome to use.

Table 4. Time in seconds for execution of benchmarks, and slowdowns with aspects woven without function inlining and without local copies of aspect variables

Benchmark	Time (Sec)	Slowdown			
	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling
beul	24.95	25.71	50.51	14.85	13.73
crni	23.56	26.77	100.45	13.26	12.78
diff	23.82	24.18	39.51	13.04	12.41
fdtd	25.23	81.78	116.18	9.67	9.45
fiff	24.50	32.25	35.12	14.34	13.97
hnor	26.12	31.69	-	13.51	13.83
mils	25.05	68.67	-	12.48	10.74
nb1d	25.19	85.75	129.13	11.62	12.84
nb3d	24.67	64.33	115.58	9.89	11.66
capr	24.78	96.59	149.02	9.50	7.96
Geometric Mean		43.23	74.05	12.07	11.77

In Table 5, we show the results of the same experiment after enabling function inlining, but without the local copies of aspect variables. The significant decrease in runtimes that result from enabling function inlining demonstrate clearly the importance of that optimization. Clearly, the function call overhead in object-oriented MATLAB is important to consider for those who are concerned about performance, as eliminating it has obtained a speedup of

[3] http://www.mathworks.com/matlabcentral/fileexchange

[4] The time is listed in seconds, and the slowdown is expressed as a factor of the original speed. The slowdown one can expect to experience with a given benchmark is expressed by the geometric mean. Aspects whose slowdowns are denoted by a "-" were not run with the particular benchmark. The reasoning for this is that does not make sense to apply all aspects to all benchmarks - for example, the unit checking aspect cannot reasonably be used with a benchmark featuring no quantities to which units can be applied units.

Table 5. Time in seconds for execution of benchmarks and slowdown with aspects woven with function inlining but without local copies of aspect variables

Benchmark	Time (Sec)	Slowdown			
	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling
beul	24.95	8.07	10.85	5.00	4.85
crni	23.56	8.07	14.34	5.25	4.67
diff	23.82	7.97	10.98	4.66	4.29
fdtd	25.23	9.94	22.25	5.52	5.39
fiff	24.50	6.24	9.88	5.71	5.29
hnor	26.12	4.23	-	4.49	3.89
mils	25.05	8.17	-	3.70	3.50
nb1d	25.19	5.63	16.68	5.74	5.65
nb3d	24.67	6.63	19.11	5.75	5.54
capr	24.78	4.99	16.00	6.13	5.99
Geometric Mean		7.02	14.26	5.14	4.84

about 5 times in the case of the unit checking aspect. As a lack of function inlining implies that the program must make function calls each time action code is to be executed, programs with a large number of pattern matches experience a much greater slowdown. Thus the type checking and unit checking aspects, which will often perform several actions for a single line of code, experience the most significant slowdowns. Similarly, accesses to aspect variables is time consuming, with all aspects performing about 3 times faster with the local copies being made. When using object-oriented MATLAB code, the time required to access object properties is significant, and any program which makes heavy use of object properties would likely see a notable performance increase through local copies.

6.3 Code Size Increase

Another factor to consider is that using aspects also results in a notable bloating of the program size after weaving. The results of our code size evaluation can be seen in 6, which shows the increase in lines of code for woven aspects over the base code of the benchmarks the aspects are woven into. All aspects result in a significant increase in code size, with the type aspect resulting in woven code that is on average 4 times as long than the original and the unit aspect resulting in woven code that is on average 14 times longer than the original. This may seem like a significant increase, but it is to be expected. Due to the fact that all action code is inlined, our aspects can end up weaving several lines of code for each line of code in the original program. In addition, simplifications which split the program into three-address code in order to allow for precise weaving also greatly expand the program size. For most applications, this significant increase in code size is likely not too great a concern.

7. Related Work

In this paper, we presented the development of an aspect-oriented compiler targeted at dynamically typed, array based scientific languages. In this section, we contrast our work with other approaches to the problems we address.

There have been very few attempts at bringing aspect-oriented techniques to scientific programming. Of note is the work by Cardoso et Al. [3], who also target the MATLAB language, and bring aspect-oriented features to it. Their approach focuses primarily on the problems of monitoring variables and tracking behaviour in embedded systems. The implementation they propose features only static pointcuts, matching function calls, variables, functions, tags, programs, and MATLAB reserved keywords. They introduce the concept of ”tags”, special comments that are acknowledged by their compiler in a similar fashion to ASPECTMATLAB annotations. These tags however, operate somewhat differently from our annotations, in that while they allow code to be woven around them, they cannot contain information which might be used by the woven code.

Another approach to enriching scientific programming using aspect-oriented techniques is demonstrated by Irwin et al. [5]. They introduce an aspect-oriented approach to handling sparse matrix code. Their approach allows for a user to write high level matrix code, but annotate it with information regarding an efficient implementation. As a base language, they approximate the MATLAB language, but allow additional information to be passed regarding the data representation of matrices. This information is then interpreted by an aspect weaver, which weaves the appropriate data structure into the base code, which is then compiled into C/C++.

A similar approach for handling the dynamic types in MATLAB using an aspect-orientation, which inspired our current approach, has been documented in [4]. It proposes the use of “atype” statements, which provide similar type information about variables as we require in our type annotations. The “atype” approach was a paper design and did not have an implementation. Another significant difference between this proposed strategy and the one we have adopted is the placement of annotations inside of MATLAB comments. The tradeoff here is that by placing type information in comments, it is possible to run annotated code without first using the ASPECTMATLAB compiler, however by using atype statements within MATLAB code, it would be possible to have an aspect which matches these statements, as well as a MATLAB function which executes the necessary type checking code even if aspect code is not woven - at some performance cost. For our implementation, we have decided that the best strategy would be to ensure that it is possible to run code without checks or performance loss.

Other approaches to types in dynamically typed scripting languages have been demonstrated by Ren et al. [13] , who introduce rtc, an annotation-based dynamic type checker for Ruby. Their design goals are similar to our own, in that they allow for checking types only when required, to support rapid prototyping and provide flexibility to the programmer. Their approach differs from our own in that it exists a Ruby library with methods for type checking on objects, as opposed to using a separate compilation process.

In AspectJ, it is possible for patterns to match on constructs based on the annotations they are annotated with. Note, this is different from MATLAB, where our annotations do not annotate constructs, and instead designate important areas in code. Noguera et al. [11], introduce the idea of dynamic annotations to AspectJ. These achieve a similar goal to what we have done with ASPECTMATLAB annotations, though are presented in a very different context. Their implementation allows for annotations to communicate with the aspect compiler by associating them with conditions that determine when they should be active. This is somewhat limiting, as the communication is limited to whether the annotation is either in an on or off state, but succeeds at providing additional flexibility to users of AspectJ.

In our paper, we provide an aspect-oriented approach to dimensional analysis in MATLAB. Using units of measurement as types has also been implemented in other languages, such as F# [7]. Their implementation allows for measure constructors, which can be used to parameterize other types. The end result is similar to what we have achieved, with units being automatically updated and checked after each calculation. Other approaches to handling dimensional analysis have been done in with the Osprey project [6] by Jiang et al. They introduce a constraint based approach which models units as types. They use annotations on variables, which contain similar information to our unit annotations, and use these annotations to statically detect and report errors.

Table 6. Size in lines of code of benchmarks and increase after weaving aspects

Benchmark	LOC	Increase in LOC					
	Original Program	Type Checking	Unit Checking	Dimension Profiling	Integer Profiling	Loop Unrolling	Loop Reversal
beul	23	4.30	10.82	2.10	2.08	-	-
crni	194	3.89	12.15	2.24	2.26	42.12	3.49
diff	115	4.76	9.12	2.13	2.05	46.22	4.21
fdtd	77	5.20	43.19	1.98	2.13	37.28	-
fiff	105	3.42	16.84	2.38	2.34	31.81	4.12
hnor	22	3.13	-	2.27	2.00	44.63	2.45
mils	31	2.18	-	1.87	1.94	-	-
nb1d	166	4.95	12.36	2.34	2.37	57.16	-
nb3d	141	5.79	13.54	2.62	2.51	28.12	-
capr	206	4.30	12.07	2.56	2.24	15.23	-
Geometric Mean		4.03	14.66	2.24	2.19	40.06	3.49

8. Conclusions and Future Work

In this paper, we have provided several valuable extensions to the AspectMatlab language. We have carried out various optimizations to improve the performance of aspect-woven code, and introduced several new aspects with the intent of helping Matlab programmers better understand and use their programs.

The introduction of the annotation pattern allows for programmers to communicate with aspect code through annotations, enabling an exchange of relevant information. The type and dimension patterns allow programmers to further specify conditions under which they expect their action code to be woven. The new body keyword for around advice on loops allows more flexibility for programmers who wish to take advantage of loop patterns.

The introduction of new aspects makes it easy for programmers to get started using AspectMatlab. By introducing a variety of new aspects, it is possible for novice programmers to gain immediate benefits from using AspectMatlab, even without a full understanding of the language itself. Our type and unit checking aspects allow for Matlab users to better communicate the intentions of their code, and provide them with tools to verify that their code operates as expected. Our profiling aspects also extend programmers' ability to understand their code. The loop transformation aspects we have included make it easier for programmers to find ways in which they can optimize their code.

By implementing several optimizations to the woven AspectMatlab code, we significantly improve performance of aspect-oriented code. These optimizations have been demonstrated to have a significant impact, and expected performance gains have been shown to easily amount to an order of magnitude.

Our future work extends in several directions. The language itself holds great potential for extension, and new patterns can be created to target Matlab language constructs that we do not currently handle.

Of utmost importance is the development of more general-purpose aspects which would also contribute to the utility of AspectMatlab. By packaging the compiler with a greater number of aspects that easily allow scientists to gain advantages with their Matlab programming, there will be an enhanced incentive to use the compiler. Performance improvement is another area of our future work. While our experience with inlining action code has shown it to be beneficial in most cases, it is possible that a more selective heuristic for determining when function inlining should be performed could lead to further performance increases. Finally, knowledge obtained from type annotations as we have described could also be used by a Matlab JIT compiler to produce more efficient code.

The AspectMatlab compiler and example aspects described in this paper can be found at `http://www.sable.mcgill.ca/mclab/projects/aspectmatlab/`.

References

[1] T. Aslam. AspectMatlab: An Aspect-Oriented Scientific Programming Language. Master's thesis, McGill University, 2010. URL `http://www.sable.mcgill.ca/mclab/aspectmatlab/index.html`.

[2] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, March 2010.

[3] J. a. M. P. Cardoso, J. a. M. Fernandes, M. P. Monteiro, T. Carvalho, and R. Nobre. Enriching MATLAB with aspect-oriented features for developing embedded systems. *J. Syst. Archit.*, 59(7):412–428, Aug. 2013. ISSN 1383-7621. . URL `http://dx.doi.org/10.1016/j.sysarc.2013.04.003`.

[4] L. Hendren. Typing aspects for MATLAB. In *Proceedings of the Sixth Annual Workshop on Domain-specific Aspect Languages*, DSAL '11, pages 13–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0648-5. . URL `http://doi.acm.org.proxy1.library.mcgill.ca/10.1145/1960496.1960501`.

[5] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 249–256. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63827-8. . URL `http://dx.doi.org/10.1007/3-540-63827-X_68`.

[6] L. Jiang and Z. Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 262–271, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. . URL `http://doi.acm.org.proxy1.library.mcgill.ca/10.1145/1134285.1134323`.

[7] A. Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, CEFP'09, pages 268–305, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17684-4, 978-3-642-17684-5. URL `http://dl.acm.org.proxy2.library.mcgill.ca/citation.cfm?id=1939128.1939136`.

[8] MathWorks. *MATLAB Programming Fundamentals*. The MathWorks, Inc., 2009.

[9] C. Moler. The Growth of MATLAB and The MathWorks over Two Decades, . `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.

[10] C. Moler. The Origins of MATLAB, . `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html`.

[11] C. Noguera, A. Kellens, D. Deridder, and T. D'Hondt. Tackling pointcut fragility with dynamic annotations. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0536-5. . URL `http://doi.acm.org.proxy1.library.mcgill.ca/10.1145/1890683.1890684`.

[12] M. J. Quinn, A. Malishevsky, and N. Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, pages 114–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8579-4. URL `http://dl.acm.org.proxy1.library.mcgill.ca/citation.cfm?id=822083.823199`.

[13] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The Ruby Type Checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1565–1572, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1656-9. . URL `http://doi.acm.org.proxy1.library.mcgill.ca/10.1145/2480362.2480655`.

[14] L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2): 286–323, 1999. ISSN 0164-0925. .