McCogen, A C Code Generator from Mathematica Language

Chang-Hyeon Song

Report Number:
96-054
MC COGEN, A C CODE GENERATOR
FROM MATHEMATICA LANGUAGE

Chang-Hyeon Song

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-054
September 1996
McCogen, A C Code Generator From Mathematica Language

Song, Chang-Hyeon

August 1996

Abstract

This thesis presents a program, named MeCogen (Mathematica to C automatic Code Generator), that generates C code (program) from Mathematica code.

It improves upon the slow performance of Mathematica compared to compiled languages like C. Secondly, it alleviates the need of redundant coding when users want to port their Mathematica code to C. Lastly, it might help students who know Mathematica language to learn C programming easier.

MeCogen was written in ANSI C and based on MathLink, which is the standard structured communication method in Mathematica, so it is highly portable. As a result, MeCogen is just another Mathematica installable function.

C code automatically generated by MeCogen is syntactically correct so that it compiles with no or minimal modifications, and includes correct include statements for header files, easy-to-read indentation, declarations of all variables, tight memory usage (dynamic allocation), call to existing library functions, and even symbolically manipulated expressions.

The main advantage of MeCogen over other implementations is that it can generate a whole syntax-error-free C program and can be linked with existing libraries.

1 Introduction

1.1 Objective

Mathematica enables scientists and engineers to solve simple to mid-size problems because of its fast prototyping capability. Since numerous symbolic and numeric built-in or package functions are available in Mathematica, it is easy to formulate problems and get rough solutions quickly. However, scientists and engineers are reluctant to solve large scale or even mid-size problems in Mathematica because of the performance bottleneck compared to that of compiled programs written in a conventional programming language like C or Fortran. This is especially true when there are many loops in Mathematica code. The performance ratio ranges from 2 to over 100 when a Mathematica code contains iterative algorithms with major loops (for example, Do, For, While). What they do to overcome this situation is to formulate the problem in Mathematica first, get a rough idea of the solution, and then write an equivalent program in a conventional programming language. With an automatic code generator like MeCogen, a great amount of coding time can be saved.

Another reason for the need of an automatic code generator is that with such a system it is easier to port or incorporate Mathematica code into bigger problem solving environments.

Suppose one has written a 3D mesh generator in Mathematica for the finite element method for solving PDE problems and wants to use the mesh generator code for parallel ELLPACK [HR92] to solve PDE problems. Without an automatic code generator or translator, one has to write an equivalent mesh generator using C or Fortran. The major reason why MeCogen will lessen the burden of porting is that it will prevent unnecessary bugs that can be introduced by human programmers when porting Mathematica code. It is most unlikely that ported C codes will compile cleanly without spending a fair amount of time on debugging. MeCogen can save one from this frustration.

Additionally, students can take full advantage of a code generator. In fact, the students in CS158A, an introductory C programming course at Purdue University, may have benefited from MeCogen. Since they were exposed to Mathematica language before they began to learn programming in C, they could make a much easier transition from Mathematica programming to C programming with the help of MeCogen.
1.2 Features of McCogen

Within Mathematica session McCogen interactively generates C code (entire program). The following is a list of main features of McCogen.

- McCogen uses MathLink as a means to communicate with the Mathematica kernel.
- It can generate C code that uses existing function libraries.
- It evaluates symbolic expressions, and uses the result instead.
- It automatically detects data types of all variables.
- It automatically includes
  - declaration of all variables.
  - header files.
  - easy-to-read indentation.

As a result, McCogen is able to generate C code that can be compiled and executed with no or minimal modifications.

1.3 Mathematica

1.3.1 Mathematica Programming Language

Mathematica, developed by Wolfram Research, Inc., is a well-known commercial system and language that can be used as a numerical and symbolic calculator, visualization tool, programming language, modeling, data analysis, interactive documents, control language, software platform (that can be used to develop software and execute on), and embedded systems (called within other systems) [Wo19]. Mathematica language is very versatile and has many different faces such as object oriented, imperative, functional, etc. A number of people have demonstrated Mathematica's versatile programming language features in their books, [GKW96], [Mae91], etc.

1.3.2 MathLink

The implementation basis of McCogen is MathLink, the Mathematica communication standard. MathLink allows a higher level communication between Mathematica and external programs and also allows external programs both to call Mathematica, and to be called by Mathematica [Wo19]. It also can be used to let Mathematica call individual functions inside an external program [Wo19]. Greater detailed description will be given in section 4, where the implementation issues are discussed.

1.4 Overview of the Thesis

What will be presented in the next section is similar systems to McCogen. Also presented along with them are how they are different from McCogen and how McCogen addresses the shortcomings of the system under consideration.

In section 3, the design strategy of McCogen will be discussed. There are many issues to consider when translating one language to another, and issues specific to C code generation from Mathematica language will be discussed in great detail.

In section 4, how McCogen system is implemented is presented. Implementation details along with sample sessions will be illustrated.

In section 5, two examples of the application of McCogen will be presented. A simple example of using McCogen will guide the user on how and where to use McCogen, and one complex application will show how to build up a complex code generation system using McCogen. Additionally presented is the performance comparison between execution speed of input Mathematica code and generated C program.

In the last section, what has to be done to improve McCogen will be discussed along with words of conclusion.


2 Related Work

There are many code generation systems, but most of them are the systems that take their own high level descriptions or languages. And they can only generate target code for specific applications such as PDE solver, which means that they cannot be used as general purpose code generating systems. Presented here are the systems that take existing and well-known Mathematical software languages like Mathematica, Maple, Matlab, or Reduce and translate them to target programming languages in an intelligent way.

There are few Mathematica to C or Fortran translators available. There are some Maple to C, Macsyma to C, Matlab to C, and Reduce to C translators, and they also are presented in this section to give a rough idea how McCagen is different from those code generation systems.

2.1 MACTRAN

Developed by Wirth [Wir80], MACTRAN provides a general framework in which Macsyma [Inc95] can be used to code subroutines. But this system is seriously limited in the sense that it can only translate Macsyma statements to FORTRAN statements, no program structure at all. Template files are needed to generate a section of FORTRAN code for all applications. Furthermore, the generated FORTRAN code will neither include variable declarations, nor provide general mechanism to reuse preexisting function libraries.

2.2 GENTRAN

GENTRAN [Gat86] is an automatic code generation package for Reduce [Cor85] system. GENTRAN under consideration here is the second version, which was implemented in RLISP [Gro84], and it translates Reduce prefix forms into formatted Fortran77 [Ins78], Raffor [Ker79], and C [KR88] code. The package is also available in Macsyma. GENTRAN is an interactive package that can be used in Reduce session to generate target code on-the-fly. It has a facility to generate target code from symbolically manipulated Reduce expressions through eval function. Additional features are delayed or greedy evaluated expressions, template processing, the segmentation of long expressions, and code optimization [Gat86].

GENTRAN is the most similar code generation package to McCagen in a sense that it will take a set of expressions in a source language and produce target code that includes a program structure with type declarations, automatic indentation, and delayed and greedy evaluated expressions.

However, to use GENTRAN facility users have to modify or start with new Reduce or Macsyma code, because there are separate operators and functions for GENTRAN. Besides GENTRAN cannot generate C code that utilizes preexisting function libraries.

But McCagen addresses both of these problems: users do not have to modify existing Mathematica code, and they can use preexisting libraries.

2.3 Sofroniou’s Mathematica Package

This is a Mathematica package that extends built-in format rules [Sof93]. The package takes advantage of Mathematica’s Splice command and can be used to generate C, Fortran77, Maple, and TeX [Knu89] code.

The package is limited in its functionality as a code generator in a sense that it can only translate expressions; no automatic type declaration is performed. Furthermore, users have to modify existing Mathematica code or start from a new code. And this package doesn’t generate C code that calls existing library functions, either. But it addresses some important issues, such as strict adherence to IEEE precision of numbers and expression optimization.

2.4 MATLAB Compiler

Matlab is a commercial program, developed by MathWorks, that can be used to solve matrix oriented problems in science and engineering fields. The MATLAB compiler, also developed by MathWorks for Matlab, translates M-files, which contains Matlab functions, into C code. It can generate stand-alone C code, but the code has to be compiled with MATLAB C Math library. Both of them, MATLAB compiler and the MATLAB C Math library, are commercial products unlike other systems this section describes.
The MATLAB compiler has two major drawbacks; it can translate only MATLAB functions not scripts, and global variables are not automatically declared (no type declaration). Furthermore, users cannot use their own function libraries either.

3 DESIGN of MCCOGEN

Before the code generation process is described, main difficulties regarding C code generation from Mathematica language are discussed in this section.

Because of the feature-richness and interpretive nature of Mathematica language, it is hardly feasible to convert one statement after another.

The main difficulty of generating C code comes from the fact that all Mathematica data objects or variables do not need to be type-declared, while all variables in C have to be explicitly type-declared.

Additionally, it is important to keep the semantics of Mathematica language because there are differences, sometimes subtle, between C and Mathematica operators even though the same symbol is used.

Furthermore, when a Mathematica code contains some symbolic functions such as differentiation or integration, it is hard to determine what to do with them because there are no corresponding symbolic C functions. Translating a symbolic function like \texttt{Integrate} to a C function directly is out of the question.

As a result, some requirements have to be imposed on Mathematica code so that the corresponding C code can be successfully generated.

3.1 Type Inference Problem

In Mathematica all variables are randomly typed, which means that Mathematica doesn't require type declaration of variables before their usage. So users can assign any type of data to variables no matter what types of data they were assigned previously. On the other hand, variables in C can only store specified byte-sized data types as they were declared. For example, suppose 3 was assigned a variable \texttt{a}. It can also be assigned a string later. But in C, this assignment to a string is not allowed if the symbol \texttt{a} was declared as an integer type.

3.1.1 Declaration of Variables

Since Mathematica does not require variables to be type-declared prior to the assignments, there is no way to tell if the variables will be assigned certain types later in the code. Therefore, McCogen has to look at the entire Mathematica code to determine which data types the variables should be declared as. This requirement leads to one important issue of implementation of McCogen: McCogen needs entire input Mathematica code in memory while and even before code translating takes place. This is another reason why McCogen cannot translate one statement after another.

The data types of variables are determined solely by assignment statements, which is \texttt{Set} or \texttt{=}. The final data types for variables will be the biggest data types that have been assigned to them. In this way generated C code will use memory tighter than C code that declares all variables as \texttt{double} or \texttt{long double}.

There are subtle differences in assignment statements between C and Mathematica. In C, the following assignment statements \texttt{a=b=c=1} will be interpreted as 1 is assigned to \texttt{c}, and the content of \texttt{c} is assigned to \texttt{b}, and the content of \texttt{b} is assigned to \texttt{a}. What Mathematica does, however, is to assign 1 to \texttt{a}, \texttt{b}, and \texttt{c}. So even if \texttt{b} was assigned a real number previously, \texttt{a} shouldn't be typed real to keep the semantics of Mathematica assignment, \texttt{=} . \texttt{a} should be declared as an integer variable since it is assigned a value 1 unless it was assigned different data type.

As an example, consider the following assignments.

\begin{verbatim}
b=c;
\end{verbatim}

Since \texttt{c} has not been defined yet, \texttt{b} has to be a pointer variable (unknown data type) that points to \texttt{c}.

\begin{verbatim}
a=b;
\end{verbatim}
also has to be a pointer variable with unknown data type since \( a \) is a pointer variable, but \( a \) does not point to \( b \), but points to where \( b \) points to, which is \( c \). It is because \( b \) still has not assigned any data value yet.

\[ c = 5; \]

After this assignment, \( c \) can be declared as an integer variable, and both \( a \) and \( b \) can be declared as integer pointer variables.

\[ b = 4; \]

\( b \) was determined to be an integer pointer variable. Now it is assigned an integer value. In C, this will screw up pointer value with \( b = 4 \); but if \( *b = 4 \) is executed instead, \( c \) should get a new integer value since \( b \) is pointing to \( c \). However the same analogy cannot be applied for Mathematica assignments; \( b \) should become integer type from integer pointer type if the semantics of Mathematica is kept. This problem renders McCogen to impose restrictions on assignment statements.

To overcome this problem of selecting a data type for each variable, non-castable mapping should not be allowed. That is when symbol \( a \), which has previously been assigned an integer, is now assigned a real number, the symbol \( a \) can be declared as float or double. It is allowed because an integer can be casted to a float and vice versa although it is not a good idea to cast a float to an integer. But when the symbol \( a \) is assigned a string, and a string cannot be casted to an integer in C, the code generation will fail.

Another restriction is that once a variable is determined to be an undefined pointer type, no assignment is allowed.

As described earlier, neither \( b = 4 \) nor \( *b = 4 \) is appropriate.

The steps to take to identify variables' types are the following.

1. Through the first assignment statement to a variable, McCogen records, internally, its type by looking at the right hand side of the assignment.
   - If right hand side is a variable and undefined yet, the variable on the left hand side is declared as an undefined pointer type that points to the variable on the right hand side.
   - If right hand side is a variable and its type is already determined, the type is used to determine the data type of the variable on the left hand side.
   - If right hand side is a scalar value such as integer or real, the data type is recorded. The recorded data type is used to determine data type of the variable on the left hand side.

2. To determine the data type of a variable, the following guideline is used.
   - If the data type of the current variable is undefined, then the type of the right hand side is set as its type.
   - If the data type of the current variable is defined already McCogen will compare the size, actual byte size required for each C data type, of the right hand side of the assignment and the currently recorded data type. Then McCogen sets the bigger data type for the variable.

3. If the variable above is used again in the left hand side of an assignment statement, the above procedure is used again to determine the data type.

4. Once data type for a variable is determined, update data types of all pointer variables that point to the variables.

The above procedure is summarized in the diagram in Figure 1.
3.1.2 Return Type of Functions

In C, input parameters and return type of functions are explicitly typed. But in Mathematica, neither input parameters nor return type are required to be explicitly typed. Furthermore, the input parameters of Mathematica functions are not typed because if input parameters are typed the functions will be defined only for the specific type of input parameters so that some symbolic functions would not work.

However, it is not possible to determine data type for input parameters and return value unless, at least, input parameters are explicitly typed. This requirement for explicit data typing of input parameters also enables McCogen to determine types of local variables of functions. To specify data type of input parameters in Mathematica, the head of each input parameter has to be specified. For example, function \( f[x] := 2x \) should be modified to \( f[x_{\text{Integer}}] := 2x \).

But sometimes even though input parameters are explicitly typed, the return type may be different from what Mathematica function should return.

Consider a function

\[
f[x_{\text{Integer}}, y_{\text{Integer}}] := x/y
\]

Syntactically \( f \) could return integer type since all input parameters are integers. And Mathematica handles arbitrary precision arithmetic \( f \) by always returning the correct result of a division. But in C, only one data type can be returned as a result of division. Suppose \( f \) returns an integer. If \( x \) divides \( y \) exactly, then there is no problem. But if \( x \) doesn’t divide \( y \) exactly, in other words, Mathematica function \( f \) returns a rational or real number, the result will be casted to an integer value in C; this leads to a wrong result of the division. For example, upon executing \( a = f[1, 2] \), \( a \) should store 0.5 in Mathematica. But if the corresponding C function \( f \) returns an integer type, the result would be 0. This is not how Mathematica behaves; the Mathematica function should return 0.5.

To avoid this problem it is necessary to return a floating point number (float, double, or long double) even if all variables associated with the division operator are integers.
3.2 Precedence Rules

3.2.1 Representation of Expressions

Prior to the discussion of precedence rule issue, it is important to know how Mathematica keeps its expressions internally.

Mathematica expressions can be either atoms or composite, which is combined expressions of atoms. Atoms are numbers, strings, and symbols; composite expressions are expressions that are built from one or more expressions of the prefix form Head[arg1, arg2, ..., argn], and head itself is a Mathematica expression [Res93]. For example, a=1 is represented internally as Set[a, 1]. The representation can be easily shown by giving an expression to argument to FullForm[HoldForm[ ]]. For example.

```
FullForm[HoldForm[a=1+1/3;a=Abs[a]]]
```

will return

```
CompoundExpression[Set[a, Plus[1, Times[1, Power[3, -1]]]], Set[a, Abs[a]]]
```

3.2.2 Automatic Application of Precedence Rules to C

When converting one language to another, keeping precedence rule of the original language in target language is essential [Sof93]. Without careful parenthesizing of expressions it is easy to produce unanticipated expressions in the target language. Very fortunately there can be no confusion on how to apply target language's precedence rule thanks to MathLink. Through MathLink, which is the implementation basis of McCogen, all expressions communicating between a Mathematica kernel and McCogen are in FullForm. To be exact, given Mathematica code is automatically parsed by Mathematica and Mathematica generates strictly formatted expressions. Parentheses are used as much as possible to keep its original precedence.

3.3 Scope Rules

Mathematica supports two types of scope rule in the functions (SetDelayed): lexical and dynamic. Lexical scope in Mathematica works exactly like the scope rule in C functions, in which all the variables are either local or global. But the dynamic scope rule is not supported in C, in which variables belong to the local context of execution history.

The Module mechanism implements the lexical scope rule, while Block implements the dynamic scope rule. Since dynamic scope is not supported in C, Block cannot be used to generate C code.

Nevertheless the lack of dynamic scope rule is not necessarily a serious problem as far as programming computational science applications is concerned since few programs make use of dynamic scope rule.

3.4 Supported Data Structures

As far as basic data structure is concerned, scalars, vectors, and two dimensional matrices currently are supported. For vectors and matrices, size or rows and columns are kept in corresponding variables (.size appended for vector; .row and .col appended for matrices) since there is no way of knowing the size of vectors or matrices after malloc or calloc. An array variable is declared as a pointer rather than an array in C because pointer variable allows resizing the memory allocated unlike array, in which users have to declare an array of new size and copy the entire content of old array over to the new array.
3.4.1 Lists

Vectors and multi-dimensional matrices can be represented as lists or lists of lists in Mathematica. Mathematica lists can contain any types of data as opposed to C arrays, which can only contain data of a single type. It is not practical to allow multiple types in a list when converting to a C data structure for several reasons. One reason is that C does not allow multiple types in the array structure. Another reason is that if list is implemented as a linked list in C, additional information, such as type of data and pointers to next nodes, have to be included in each data (node), which will double the memory requirement easily; besides it is seldom practical to use linked lists as matrices for computational efficiency because locality of reference will be destroyed thanks to the pointers used in nodes of linked lists.

To identify the size of vectors or matrices McCogen has to look at their initialization. The initialization can be done in the form of

\[
\begin{align*}
\text{vector} &= \{1, 2, 3, 4, 5\} \\
\text{matrix} &= \{(1, 2, 3), (4, 3, 2), (5, 2, 4)\}
\end{align*}
\]

Alternatively, users can use `Table` function to initialize vectors and matrices. For example,

\[
\begin{align*}
\text{matrix} &= \text{Table}[0, \{200\}, \{300\}]
\end{align*}
\]

will initialize a matrix of size 200 by 300 filled with 0.

Mathematica supports basic arithmetic operations such as addition or multiplication of vectors or matrices. In C, those arithmetic operations can only be applied to scalars, which means that users have to provide their own functions or write appropriate C code for the operations on vectors and matrices. It is possible to replace some basic arithmetic operations with appropriate C code upon translation, but it is better to leave it to users. Because it could be that the matrices are in sparse form so that default function for matrix multiplication cannot be applied.

As a result, users have to add C code for the basic arithmetic operation such as multiplication or addition of vectors and matrices into the generated C code.

3.5 Symbolic Computation

There is no corresponding intrinsic, ANSI library, symbolic C functions with respect to Mathematica symbolic functions like `Integrate`. So when input Mathematica code includes symbolic functions C code generation will fail. It is possible to use some numerical function libraries that implement symbolic functions, but the number of functions that can be recognized would be severely limited. Besides using a specific numerical function library is not the best solution for translating Mathematica symbolic functions.

However McCogen will translate most symbolic functions and put the evaluated expressions as C code; in other words, McCogen forces Mathematica to evaluate the symbolic functions and tries to translate the evaluated expressions. Detailed descriptions on how this is done will be given in Section 4.

The following is a sample session that shows how the symbolic functions will be translated into C code.

\[
\begin{align*}
\text{In}[5]:= \\
\text{M2C}\{x=1, y=2, \\
a &= \text{Integrate}[x^2 + x \cdot y, x], \\
b &= \text{D}[\text{Integrate}[	ext{Tan}[x], x], x] + \text{Sum}[1/x, \{x, 10, 100\}], \\
c &= \text{Normal}\{\text{Series}[\text{Exp}[x], (x, 0, 5)]\}, \\
d &= \text{Limit}[\text{Sin}[x]/x, x\to0]\}
\end{align*}
\]
```c
#include <math.h>

/* Global variables */
int x, y;
double a, a, b, c, d;

int main()
{
    x = 1;
    y = 2;
    a = (0.5 * pow(x, 2)) + (0.3333333333 * pow(x, 3)) + (x * y);
    e = 383.3333333;
    b = ((cos (log (cos (x)))) * tan (x)) + 2.358409264;
    c = (1 + x + (0.5 * pow(x, 2)) + (0.1666666667 * pow(x, 3)) +
        (0.0416666667 * pow(x, 4)) + (0.008333333333 * pow(x, 5)));
    d = 1;
}
```

### 3.5.1 Recursive Evaluation

When McCogen makes Mathematica evaluate symbolic expressions, McCogen will try to recursively evaluate the expressions returned from Mathematica as results of the evaluation of previous symbolic expressions until Mathematica's internal iteration limit is reached. For example, if Mathematica evaluates

```plaintext
Integrate[Cos[x]/Sin[x], x]
```

it will return

```plaintext
Log[Sin[x]]
```

which will be recognized by McCogen, and C expression \( \log{\sin{(x)}} \) will be inserted. But if Mathematica evaluates

```plaintext
Integrate[Cos[x^2] Sin[x], x]
```

, it will return itself, which is

```plaintext
Integrate[Cos[x^2] Sin[x], x]
```

as a result. McCogen obviously doesn't recognize \( \text{Integrate} \) again, so McCogen will make Mathematica evaluate the result again; and McCogen does it recursively. Ultimately Mathematica's iteration limit will be reached, and the code generation will fail. As a result, it is the users' responsibility to make sure that all returned expressions do not include any symbolic expressions that cannot be resolved later in the recursive evaluations.
3.5.2 Problems with User Functions

Since the types of input arguments to user-defined functions have to be specified for successful C code generation, the application of symbolic function may not work as it should. To be exact, user-defined functions are now defined only for the given typed argument. For example, consider the following Mathematica code.

\begin{verbatim}
In[4]:= f[x_] := x^2
In[5]:= Integrate[f[x], x]
\end{verbatim}

However the argument x to function \textit{f} has to be typed for successful C code generation; then \textit{Integrate} function won't work.

\begin{verbatim}
In[9]:= g'[x]
\end{verbatim}

But this causes another problem with \textit{Derivative}. Consider the following code.

\begin{verbatim}
In[12]:= Derivative[1][x^2][x]
\end{verbatim}
The result is not what users expected. To solve the problem, \( D \) has to be used instead of \texttt{Derivative} or \texttt{'}. The following command will work.

\begin{verbatim}
In[13]:= D[x^2, x]
Out[13]= 2 x
\end{verbatim}

To make a long story short, users have to replace all \texttt{Derivative} or \texttt{'} with \texttt{D} for successful C code generation. For example,

\begin{verbatim}
In[14]:= M2C[a=1; f[x_Real] := x^2; b=D[f[a], a]]
Out[14]=
   #include <math.h>
   /* Function prototypes */
   double f (double);
   /* Global variables */
   int a;
   double b;
   double f (double x)
   {
       return ((sin(x) + cos(x)));
   }
   int main()
   {
       a = 1;
       b = (cos(a) + (-1 * sin(a)));
   }
\end{verbatim}

\section*{3.6 Using preexisting libraries}

It is important to reuse preexisting functions (libraries). McCogen allows users to use preexisting library functions in the generated C code, and this is one of the features of McCogen that distinguishes itself from other C code generators. This feature is an important part of what makes McCogen generate a C code (program) that can be compiled with no modification at all. This feature lets users use their C library functions in Mathematica code as if they were Mathematica's builtin functions. All McCogen does is to put dummy Mathematica functions whose names are the same as corresponding C library functions, and translates into the C function calls as if they were built-in Mathematica functions. To do that users have to enter C library function name, C header file where prototype resides, a list of arguments to the function, and a return data type. Detailed explanation on this issue will be covered in the next section.

\section*{3.7 Precision of Arithmetic}

\subsection*{3.7.1 Keeping Original Precision of Numbers}

This issue relates deeply to the matter of what type of data can be passed from Mathematica on implementation level. If the given Mathematica code will be given as texts (strings) directly to McCogen, it would be possible to keep original floating point precisions, but it is not the case for McCogen and it will be discussed in the next section in which implementation details are presented.
3.7.2 Arbitrary Precision Arithmetic

One of the most powerful features of Mathematica is its capability of handling arbitrary precision arithmetic. This is somewhat related to the nature of not requiring explicit data typing, since a variable can be assigned to a number of any floating point precision. Arbitrary precision arithmetic is impossible in conventional programming languages like C or Fortran without the help of third party libraries, since all variables have to be declared to carry only fixed floating point precisions.

3.8 C Code Generated by McCogen

C code McCogen generates from a given Mathematica code will be syntactically correct so that compiling the C code will be successful without any syntax error.

3.8.1 ANSI C

C code that McCogen generates is ANSI C code. It strictly follows syntax rules given in [KR88]. As a result generated C code can be compiled with any ANSI C compiler like gcc without any syntax error. There are some cases in which users should modify the generated C code; for example, matrix and vector multiplications or additions. As described earlier the code for the arithmetic operations for vectors or matrices is not generated automatically so users have to modify or add C code for the operations.

3.8.2 Headers

Header files that generated C code uses will be inserted in the beginning of the code automatically. For example, if there is a `printf`, which is translated to `printf` in C, then `#include `<stdio.h> will be added automatically. If any of the mathematical functions like `sin` is used `#include `<math.h> is added, too.

3.8.3 Indentation

For readability, the easy-to-read indentation of C code is done automatically. The indentation style may be similar to that of K&R [SF96]. The current indent level is four spaces.

3.8.4 Function Prototypes

Function prototypes are added automatically right after inclusion of header files. These prototypes are used to check the correctness of the return type and types of input parameters in the generated C code.

3.8.5 Declaration of Variables

All variables used in given Mathematica code will be declared as global variables, and the declarations are inserted automatically right after function prototypes. And all variables used in function declarations are declared as local variables that belong to the functions.

3.8.6 Comments

All comments users put between (`*` and `*`) in Mathematica code will be removed while the expressions are converted to Full Form. If users want to add comments in generated C codes, they need to put comments with a dummy function `Comment`. Given string will be placed within `/` and `*/`. An example usage is followed.

```plaintext
In[31]:= M2C[
x=1; Comment["This is a sample comment"];
Do[Comment["Increment x"]]: x++; Comment["Increment Done"], {10}];
Comment["End of program"]
```

12
3.9 Current Shortcomings

3.9.1 Overflow

Consider a function $f(x, y) := x + y$. After generating C code from this we have the following code section.

```c
int f(int x, int y)
{
    return (x + y);
}
```

In the C code, $f(30, 435)$ will return a correct integer result. But $f(MAX\_INT/2 + 100, MAX\_INT/2 + 100)$ will not return a correct result, while corresponding Mathematica code will return a correct real number. This is because an overflow has occurred in the C function. There is no way of telling whether the operation or assignment ended up overflowed. Furthermore, it is not feasible to store all computed results, in other words, keeping the state of execution of given Mathematica code while translating. However it is not feasible to upgrade all variables to long int or long double to avoid this problem at all. So it is also users' responsibilities to prevent overflow situation.

3.9.2 Function within a Function

In C, a function cannot be defined within another function. If a Mathematica function defines another function within itself, then code generation simply will fail. Simple solution to this problem is to define the function, which is defined inside, outside of the function. However a function $A$ is defined inside a function $B$, function $A$ should not to be called from outside the function $B$, but if the function is defined outside it can be called from other functions. This is obviously not an intended side effect.

3.9.3 Mutual Function Calls

This type of function call is allowed in C with the help of function prototypes since before the actual definition of functions C compilers know what the return type and argument types are from the prototypes. When translating from Mathematica language, McCogen has to determine functions' return type before it can proceed.
For example, consider the following code section.

```mathematica
f[x_Integer] := If[x==0, x, x + g[x-1]]
g[x_Integer] := If[x==0, x, x + f[x-1]]
Print="f is ", f[5];
Print="g is ", g[6];
```

It certainly is valid Mathematica code and works well when the given argument to functions f and g are larger than or equal to zero. The desired C code from the above Mathematica program should be:

```c
#include <stdio.h>

int f(int); /* Function prototypes */
int g(int);

int f(int x)
{
    if (x==0)
        return x;
    else
        return (x + g(x-1));
}

int g(int x)
{
    if (x==0)
        return x;
    else
        return (x + f(x-1));
}

main()
{
    printf("f is ", f(5));
    printf("g is ", g(6));
}
```

However McCogen cannot get the return type of function g when it tries to translate function f. To be exact McCogen cannot get the return type for function f. As a result code generation will fail. Simple solution to this situation is to define a function twice because McCogen simply overrides the previous definition of the function. For example, user can define the function g before the function f is defined.

4 Implementation

The actual C program generator, which is named M2C, is an installable function in Mathematica, and it, of course, uses MathLink 2.2 for communication to and from a Mathematica kernel. So users can use M2C as just another Mathematica function interactively.

4.1 Installable Function: MathLink

MathLink, according to MathLink reference [Res93], is a general mechanism for exchanging mathematical expressions between Mathematica and other programs. Mathematica communicates with all external programs through MathLink. For example, Mathematica's notebook frontend for X-window is a MathLink application.
4.1.1 Why MathLink?

There are several other ways to implement C code generators such as using Splice function, which M. Sofroniou [Sof93] already did, but using MathLink has several advantages over other implementations.

- There is almost no Mathematica programming involved with the code generator itself. This isn’t necessarily an advantage to Mathematica experts, though.
- An installable function can reside on other hosts. This is an advantage against Mathematica package, which has to be in some directory that Mathematica can access (need a local copy).
- The code generator can work without a Mathematica kernel. If an application generates and writes Mathematica expressions in the form of internal expression tree of McCogen, then C code can be generated from it independently.
- It is easy to implement a mechanism to use preexisting library calls.

Using MathLink, of course, has some disadvantages.

- An installable program can be quite complex due to the fact that MathLink requires strict data typing on all information exchanged. [Sof93]
- It is not easy to keep the execution state of Mathematica code.
- Code generators have to be compiled, so users need C compilers; besides recompiling is necessary to add more features unlike Mathematica package.

Even though there are several disadvantages with using MathLink, the second advantage above alone is worth the price. In fact, CS158A students can use McCogen from their Windows NT machines even though they do not have copies of McCogen on their machines. If they have to have individual copies of McCogen they need to bring the source codes and compile it before using it. Besides the source codes of McCogen are constantly changing because of its development stage, their copies of McCogen would seldom be up-to-date.

4.1.2 Installable Functions

An installable function in Mathematica works just like any other intrinsic Mathematica functions, but the actual function that does the work is an external program that was compiled separately. The communication between a Mathematica kernel and an installable function is done by MathLink. The reason why it is called an installable function is that the function has to be installed by Install command [Gay93]. Once an installable function is installed, the external program becomes a process that runs until users issue Uninstall command.

Mapping from a Mathematica function to an external installable function is done by a template file. The template file has to be processed and compiled with the associated installable function. What has to be specified in the template file for the associated installable function are the name of the actual C function that does the work, a corresponding Mathematica function name, input parameters and their data types, and return data type. Through this template file along with MathLink, Mathematica knows how to call the external function and how to communicate with it.

The following is the template file for M2C.

```mathematica
:Evaluate: BeginPackage["ConvertMathPackage"]

:Evaluate: SetAttributes[CallPacket, HoldRest]
:Evaluate: SetAttributes[M2C, HoldAll]

:Evaluate: M2C::usage = "M2C[exp] will translate given Mathematica an expression or
expressions into a corresponding C program"

:Evaluate: Begin["Private"]
```
4.1.3 Communication

MathLink will open up a TCP, pipe, or PPC (on Macintosh) connection between an installable function and a Mathematica kernel. Normally the `Install` command will open up a pipe connection on UNIX platforms. For example, the following command will open up a pipe connection.

```
link = Install["m2c"]
```

where `m2c` is the name of an executable that includes an installable function in the current directory.

Since users can also open up TCP connections, it is possible to call installable functions that resides on other hosts or machines. For example, the following command will open a TCP connection between a Mathematica kernel and `m2c` program, which is on host `aristotle.cs.purdue.edu`.

```
link = Install["3000@aristotle.cs.purdue.edu", LinkMode -> Connect, LinkProtocol -> TCP]
```

4.2 Code Generation Process

This section presents how the code generation is done internally, and describes each step in detail. The outline of the process is illustrated in Figure 2.

4.2.1 MathLink packets

Two major tasks MathLink carries out are converting Mathematica fullformed expressions into packets that can be recognized by MathLink itself, and sending and receiving those packets to and from an installable function. Expressions that users want to translate should be given as an argument to `M2C`. For example, `M2C[ a=1; a++; Print[a] ]`.

At first Mathematica will convert the entire expressions into FullForm. To be more exact Mathematica will hold the entire expressions unevaluated, which is done by HoldForm, parses the unevaluated expressions and turns them into FullForm. Then MathLink takes the FullForm expressions, converts them into MathLink packets, and starts sending out the packets to an installable function. Mathematica will wait for the installed function to send some packets back as a return packet. Upon receiving the packet, Mathematica displays the content of the packet into its output cell, and then waits for another input.
4.2.2 An Internal Expression Tree

Upon receiving those packets from Mathematica, McCogen keeps putting pieces into an internal expression tree. The FullForm expressions will be divided into function, symbol, integer, real, and string type.

The whole expressions given as an argument to M2C is kept in the internal expression tree. The expression tree is used to generate corresponding C program, and it is the sole information McCogen uses for C code generation. The data structures for the expression tree are presented below. The reason why the data structures are important is that McCogen will generate C code only from this expression tree, so it is possible to generate C code automatically even without Mathematica. As a result, it is possible to use McCogen as a module to a bigger problem solving environment. The only exception to this is translating symbolic functions. A C program cannot be generated without the help of Mathematica kernel, so an expression tree alone will not be enough if it contains symbolic functions.

The following C codes are the data structures for internal expression tree.

```c
enum variable
{
    ERROR = 0,
    UNDEFINED,
    ANY,
    SHORT_TYPE,
    INTEGER_TYPE,
    REAL_TYPE,
    LONG_TYPE,
    DOUBLE_TYPE,
    LONG_DOUBLE_TYPE,
    STRING_TYPE,
    SYMBOL_TYPE,
    PREDEFINED_TYPE, // internal math symbols like Pi, E
    FUNC_TYPE
};

enum major
{
    UNDEFINED_TYPE,
    ANY_TYPE,
    SCALAR_TYPE,
    VECTOR_TYPE,
    MATRIX_TYPE
};
```
typedef struct
{
  enum major matype;   // Major type; scalar, vector, matrix
  enum variable mitype; // Minor type; int, float, double, ...
} var_type;

typedef struct
{
  var_type type;
  void *atom;     // Any node item; int, real, string, etc
} node;

typedef struct
{
  char* name;     // actual name of the symbol
  node* content;
} symbol_node;

typedef struct
{
  long args;     // Specify number of arguments
  node* head;
  node** child;  // array to void pointers(func or child)
} func_node;

A sample abstract diagram of the data structure of expression tree is given in Figure 3.

4.2.3 Translation

Once all expressions from Mathematica are stored in an expression tree, M2C recursively traverses the expression tree and generates C code. C code generated greatly depends and varies from functions to functions, and how each Mathematica function is translated will be illustrated in Appendix 7. Most functions like Sin, Cos, etc have their corresponding counterparts in C, but some of functions like Table, Module, etc do not have their counter parts in C, and the handling of those functions is done in a special way. Nevertheless, all functions except symbolic functions can be translated into C code just by traversing the expression tree.

As described in section 3, McCogen can translate Mathematica symbolic functions to C code. Figure 4 shows abstractly how the symbolic functions will be evaluated and translated.

The following scheme is used to resolve unknown symbolic functions.

1. As McCogen traverses the expression tree,
   - For each function node, identified by its name, McCogen branches to corresponding functions which are used to manipulate the functions. Subtree manipulation is taken care of by the function.
   - An unknown function like Integrate will make McCogen stop translating. At that point McCogen prepares appropriate packets to send its subtree whose root is the unknown function Integrate.

2. McCogen sends out the prepared packets through MathLink to a Mathematica kernel and waits for the evaluated result from the kernel.

3. Upon receiving the result, McCogen puts the result in a new expression tree whose contents will be translated instead of the the previous unknown function.

4. McCogen goes through the same procedure as above for the new tree just build.
Figure 3: Sample expression tree that McCogen uses as a sole information for generating C programs.
4.2.4 Sending Results Back to Mathematica

The translated C code is appended to an output buffer set up by McCogen as strings while McCogen executes corresponding functions by traversing the expression tree.

The final C code will be returned as a big string to the output cell through MathLink. When an error occurs while translating, an appropriate error message is written to an output buffer instead, and it will be returned to the kernel through MathLink.

So users can use a mouse to paste the generated C code, which is in the output cell, into other editors or use the following command to output directly to a file.

```c
OutputForm[
M2C[
]
] >> output.c
```

output.c can be replaced by a file name a user wants.

4.3 Using Preexisting Libraries

One important thing to point out is that the state of global variables of installable functions are kept before Uninstall command is issued. That’s because an installable function is called within the the function’s main process that includes and calls the actual function.

The fact that the global context is kept throughout the execution of an instance of an installable function gives rise to better expandability of McCogen. For instance, even without recompiling McCogen users can register functions from their libraries to Mathematica so that they can include the functions in generated C code. For example, suppose a user has written a C function harmonic() that generates harmonic number up to given integer number. Now the user wants to use it in the C code that will be generated from the user’s Mathematica code. Since McCogen doesn’t know about the C function harmonic(), the code generation will fail in the first place. But before generating C code, user can issue a special command to McCogen in order to have McCogen recognize the C function. And then later generation of C code will succeed assuming there are no other errors.
For example, the following command will make M2C recognize the harmonic() function.

```
ExistingFunction("harmonic", "my_header.h", {Integer}, {Double});
```

First argument is the name of actual C function that user wants to call in generated C code; the header file that contains prototype of the function comes next; then comes a list of types of the input arguments; then comes a list of a return type. In this example, there is one input parameter whose type is integer, and the function returns a real number. If the function does not return anything, a null list can be used instead of a list of a return type.

After this command is issued, but before Uninstall, Mathematica code can include harmonic[]. Here is an example using existing library function calls.

```
In[50]:= link=Install["m2c"]
Out[50]= LinkObject["m2c", 12, 1]
In[59]:= M2C[ExistingFunction("harmonic", "my.h", {Integer}, {Double})]
Out[59]=
    /* Global variables */
    int main()
    {
    }

In[60]:= M2C[a=1; b=harmonic[a]; c=harmonic[b*2]; Print[c]]
Out[60]=
    #include "my.h"
    #include <stdio.h>
    /* Global variables */
    int a;
    double b, c;
    int main()
    {
        a = 1;
        b = harmonic(a);
        c = harmonic((int) (b * 2));
        printf("%dn", c);
    }

In[61]:= M2C[a=2; x=harmonic[Sum[x^2,(x,0,25)]] + Sin[a Pi]]
Out[61]=
    #include "my.h"
    #include <math.h>
    /* Global variables */
    int a;
    double x;
    int main()
    {
        a = 2;
        x = (harmonic(5525) + sin(a * M_PI));
    }
```
4.4 Using M2C

It is quite simple to run M2C. First, users have to issue `Install` command to install the function. For example,

```mathematica
link = Install["m2c"]
```

The argument `Install` takes is the name of actual executable that contains the installable function. After `Install` has been issued, simply type `M2C[ ]` with input Mathematica expressions as an argument. Then it will display the generated C code in the output cell. When the users are done with M2C they can issue `Uninstall` to disconnect the link. Following is an example session of M2C.

```mathematica

e = Install["m2c"] (* Give Executable name *)
```

```mathematica
M2C[a=(({1,2,3},{3,4,5},{4,5,6}); b=(12.34, 5.543, 5.0, 5.45); i=j=0; a[[1,2]] = 4.3; Do[total = a[[i,j]] + a[[j,i]], {3}]
```

```mathematica
/* Global variables */
int a_row, a_col, b_size, i, j, for_count;
double total;
double *b; /* Vectors */
double **a; /* Matrix */

int main()
{
    a = (int **) calloc(3, sizeof( int *));
    a[0] = (int *) calloc(3, sizeof( int ));
    a[1] = (int *) calloc(3, sizeof( int ));
    a[2] = (int *) calloc(3, sizeof( int ));
    a_row = 3;
    a_col = 3;
    a[0][0] = 1;
    a[0][1] = 2;
    a[0][2] = 3;
    a[1][0] = 3;
    a[1][1] = 4;
    a[1][2] = 5;
    a[2][0] = 4;
    a[2][1] = 5;
    a[2][2] = 6;
    b = (double **) calloc(4, sizeof( double ));
    b_size = 4;
    b[0] = 12.34;
    b[1] = 5.543;
    b[2] = 5;
    b[3] = 5.45;
    i = j = 0;
    a[0][1] = 4.3;
    for(for_count=0; for_count <3; for_count++)
    {
        total = (a[i-1][j-1] + a[j-1][i-1]);
    }
```

```mathematica

/ Output
```
4.5 Inevitable Expression Optimization

Passing FullForm expressions through MathLink doesn't always lead to convenience;

- A subtraction becomes a multiplication and an addition.
  \[ a - b \rightarrow a + (b \times -1) \]

- A division becomes a power of minus one.
  \[ \frac{a}{b} \rightarrow a \times \text{power}(b, -1) \]

Inevitably McCogen has to change the mutated expressions back to their original forms.

4.6 Keeping Original Precision of Numbers

Since McCogen uses MathLink, in which all data exchanged are strictly typed, it is not always possible to keep original precision of numbers. For example, if a number from Mathematica is larger than MAXINT on one machine, it is not even possible to get the number. On Sun Sparcstation, an integer larger than MAXINT gets garbled and makes McCogen disconnected from the Mathematica kernel. The significant digits of floating point numbers are also limited to C data structures.

If numbers are exchanged in text format, it is possible to keep its original precisions of any huge numbers and strings [Gay93]. However MathLink does not send numbers in text form, and the conversion between text and numbers back and forth has to be manually done and hardly becomes automatic.

As a result, it is users' responsibility to keep numbers small enough to fit in C data structures.

4.7 Portability and Efficiency

4.7.1 Of Code Generator

McCogen is written in ANSI C based on MathLink 2.2, so it should be completely portable on most UNIX platform (original development of McCogen was done on Sun Sparcstation running SunOS 4.1.3C). There are some differences in the MathLink header files for PC, Mac, and UNIX, but all MathLink functions McCogen uses are the standard ones like MLGetInteger or MLGetFunction. One thing to note is that currently Mathematica sends all real numbers as C data type double.

4.7.2 Of Generated C code

The generated C code follows ANSI C standard as much as possible. All C functions and constants used in generated C code are defined in ANSI C libraries [HJ91].

5 Case Studies

5.1 Simple Example: Newton's Method

In this section, Newton's method to find a root of an algebraic equation is implemented in Mathematica language.
Let $r$ be a zero of function $f$ and let $x$ be an approximation to $r$. Newton's method begins with an estimate $x_0$ of $r$ and then defined inductively

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where $n \geq 0$ [KC90]. In this example, function $f(r) = -\frac{2000}{r^2} + 4\pi r$ is used. The function $f$ has a root somewhere between 5.4 to 5.5 according to Figure 5.

5.1.1 Mathematica Code

The following is a Mathematica code that implements Newton's method.

```mathematica
Clear[bPrime, FSIZE, x];
bPrime[r_] := -2000/r^2 + 4 Pi r;
FSIZE = 0.00001;
x = 4.0;
While[Abs[bPrime[x]]/N] >= FSIZE,
  x = N[x - bPrime[x] / bPrime'[x]];
  Print["The root of bPrime is r " x];
Print["The root of bPrime is r " x]
```

As described in previous section, due to the problem with symbolic differentiation, the code should be modified as following for successful C code generation.

```c
Clear[bPrime, FSIZE, x];
bPrime[r_Double] := -2000/r^2 + 4 Pi r;
FSIZE = 0.00001;
x = 4.0;
While[Abs[bPrime[x]]/N] >= FSIZE,
  x = N[x - bPrime[x] / D[bPrime[x], x]];
  Print["The root of bPrime is r " x];
Print["The root of bPrime is r " x]
```

5.1.2 Generated C code

The following is the automatically generated C code from the above Mathematica code.
#include <math.h>
#include <stdio.h>

/* Function prototypes */
double bPrime( double r);

/* Global variables */
double FSIZE, x;

double bPrime (double r)
{
    return ({{-2000 / pow(r,2)) + (4 - 3_PI * r)});
}

int main()
{
    FSIZE = 1e-05;
    x = 5;
    while(fabs(bPrime((double) x)) >= FSIZE)
    {
        x = (x + (-1 * bPrime((double) x)) / (12.56637061 + (4000
pow(x,-3))));
        printf("r = %f bPrime = %f\n", x, bPrime((double) x));
    }
    printf("The root of bPrime is r = %f\n", x);
}

5.1.3 The Result of Execution
The result of executing the above Mathematica code is following.

<table>
<thead>
<tr>
<th>r</th>
<th>bPrime</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.99555</td>
<td>-17.3654</td>
</tr>
<tr>
<td>5.38449</td>
<td>-1.31936</td>
</tr>
<tr>
<td>5.41334</td>
<td>-0.00000</td>
</tr>
</tbody>
</table>

The root of bPrime is r = 5.41334

After compiling the C code, the execution leads to the following result.

<table>
<thead>
<tr>
<th>r</th>
<th>bPrime</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.995579</td>
<td>-17.365354</td>
</tr>
<tr>
<td>5.384489</td>
<td>-1.319357</td>
</tr>
<tr>
<td>5.413237</td>
<td>-0.008447</td>
</tr>
<tr>
<td>5.419261</td>
<td>-0.000000</td>
</tr>
</tbody>
</table>

The actual root of the function f from the Newton's method was 5.41926. Both programs lead to the same result.

5.2 More Complex Application
This example illustrates how to build a bigger C code generation package for specific applications based on McCogen. The example is solving a boundary value problem of a linear elliptic partial differential equation. The PDE problem
Figure 6: Domain specification in which the PDE is defined in Example 2

This example tries to solve is in the form of

\[ L u = a u_{xx} + c u_{yy} + d u_x + e u_y + f u = g \]

where \( a, c, d, e, f, \) and \( g \) are functions of \( x \) and \( y \) [RB85].

\( L \) is a differential operator applied to \( u \), and this equation is the second order linear partial differential equation. Specifically the problem under consideration is 2D boundary value elliptic partial differential equation, where \(-ac \leq 0\). Suppose the equation is defined over a rectangular domain \( R = \{(x, y) \mid x_l < x < x_r, y_l < y < y_r\} \). Only Dirichlet boundary conditions, where no derivatives of solutions on boundaries can be specified, on four sides of the rectangular domain can be specified as functions of \( x \) and \( y \). The Figure 6 specifies the domain in which above PDE is defined; all parameters in 6 have to be defined in the Mathematica code.

In Mathematica code, each coefficient, \( a, c, d, e, f, \) and \( g \) should be defined as a function. For example,

```plaintext
a[x_Double, y_Double] := -1.0;
c[x_Double, y_Double] := -1.0;
d[x_Double, y_Double] := 0.0;
e[x_Double, y_Double] := 0.0;
f[x_Double, y_Double] := 0.0;
g[x_Double, y_Double] := 0.0;
```

The above coefficients lead to a Laplace equation, which is

\[ \nabla^2 u = 0, \]

where \( \nabla^2 \) is the two-dimensional Laplace operator defined in a rectangular Cartesian coordinates \((x, y)\) by \( \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \) [Ste70].

Additionally all parameters in Figure 6 have to be specified like

```plaintext
xMin = 0.0;
yMax = 1.0;
yMin = 0.0;
yMax = 1.0;
t[x_Real, y_Real] := 1.0;
b[x_Real, y_Real] := 0.0;
l[x_Real, y_Real] := 0.0;
r[x_Real, y_Real] := 0.0;
```

\( t, b, l, \) and \( r \), which are functions of \( x \) and \( y \), are used to specify Dirichlet boundary values for top, bottom, left, and right hand side of the rectangular domain given in Figure 6.
5.2.1 Finite Difference Method

To solve the elliptic PDE problem, a numerical method known as finite difference method, also called five-point star method, is used.

Using finite difference method, the following substitutions should be applied to its original equation.

\[ u_{xx} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \]
\[ u_{yy} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} \]
\[ u_{xy} = \frac{u_{i+1,j+1} - u_{i+1,j-1} + u_{i-1,j+1} + u_{i-1,j-1}}{4hk} \]
\[ u_x = \frac{u_{i+1,j} - u_{i,j}}{2h} \]
\[ u_y = \frac{u_{i,j+1} - u_{i,j-1}}{2k} \]

where \( h \) is grid space in \( x \) direction, and \( k \) is that of \( y \) direction.

After the substitutions, the equation in the following form can be derived.

\[ \alpha_0(i, j)u_{i,j} + \alpha_1(i, j)u_{i+1,j} + \alpha_2(i, j)u_{i,j+1} + \alpha_3(i, j)u_{i-1,j} + \alpha_4(i, j)u_{i,j-1} = g(i, j) \]

where

\[ \alpha_0(i, j) = f(i, j) - 2\frac{a(i, j)}{h^2} - 2\frac{c(i, j)}{k^2}, \]
\[ \alpha_1(i, j) = \frac{a(i, j)}{h^2} + \frac{d(i, j)}{2h}, \]
\[ \alpha_2(i, j) = \frac{c(i, j)}{k^2} + \frac{e(i, j)}{2k}, \]
\[ \alpha_3(i, j) = \frac{a(i, j)}{h^2} - \frac{d(i, j)}{2h}, \]
\[ \alpha_4(i, j) = \frac{c(i, j)}{k^2} - \frac{e(i, j)}{2k}. \]

5.2.2 Linear Iterative Solver: Gauss-Seidel

Using Gauss-Seidel method, the above equation can be solved for \( u \).

\[ u_{i,j} = \frac{g(i, j) - \alpha_1(i, j)u_{i+1,j} - \alpha_2(i, j)u_{i,j+1} - \alpha_3(i, j)u_{i-1,j} - \alpha_4(i, j)u_{i,j-1}}{\alpha_0(i, j)} \]

5.2.3 Mathematica Code

The following Mathematica code is to solve the following 2D Elliptic PDE boundary value problem.

\[ -u_{xx} - u_{yy} = \sin x + \cos y, \]

which is defined over the domain in Figure 7.

```mathematica
hdim = 11; kdim = 11; maxIter = 300;
epsilon = 0.000001;

a[x_Real, y_Real] := -1.0;
c[x_Real, y_Real] := -1.0;
d[x_Real, y_Real] := 0.0;
e[x_Real, y_Real] := 0.0;
```

27
Figure 7: Domain specification and boundary values of the PDE problem to solve

(* coefficient of u *)  f[x_Real, y_Real] := 0.0;
(* Right hand side *)  g[x_Real, y_Real] := Sin[x] Cos[y];

xMin = 0.0;
xMax = 1.0;
yMin = 0.0;
yMax = 1.0;

(* Dirichlet boundary values *)
t[x_Real, y_Real] := 1.0;
b[x_Real, y_Real] := 0.0;
l[x_Real, y_Real] := 0.0;
r[x_Real, y_Real] := 0.0;

(* Prepare a matrix *)
u = Table[0.0, {kdim}, {hdim}];

h = (xMax - xMin)/(hdim-1);
k = (yMax - yMin)/(kdim-1);

(* Boundary Values : Only Dirichlet*)
Do[y = (i-1) k + yMin; u[[i,1]] = l[xMin,y], {i, kdim}];
Do[x = (i-1) h + xMin; u[[1,i]] = b[x,yMin], {i, hdim}];
Do[y = (i-1) k + yMin; u[[kdim,i]] = r[xMax,y], {i, hdim}];
Do[x = (i-1) h + xMin; u[[i,hdim]] = t[x,yMax], {i, kdim}];

alpha0[x_Real, y_Real] := f[x,y] - (2 a[x,y] / h^2) - (2 c[x,y] / k^2);
alpha1[x_Real, y_Real] := a[x,y] / h^2 + d[x,y] / 2h;
alpha2[x_Real, y_Real] := c[x,y] / k^2 + e[x,y] / 2k;
alpha3[x_Real, y_Real] := a[x,y] / h^2 - d[x,y] / 2h;
alpha4[x_Real, y_Real] := c[x,y] / k^2 - e[x,y] / 2k;

Do[
  maxE = 0.0;
x = y = 0.0;
  Do[ x = (i-1) h + xMin; y = (j-1) k + yMin;
    lastU = u[[i,j]];
    u[[i,j]] = (g[x,y]
      - alpha1[x,y] u[[i+1,j]]
      - alpha2[x,y] u[[i,j+1]]
      - alpha3[x,y] u[[i-1,j]]
      - alpha4[x,y] u[[i,j-1]]
    )/ alpha0[x,y];
    difU = Abs[lastU - u[[i,j]]];
    If[maxE < difU, maxE = difU]
    , {j, 2, hdim-1, 1}
  , {i, 2, kdim-1, 1}
  ];

  If[maxE < epsilon, Print["Converged on iteration ", m]; Break[]]
5.2.4 Generated C Code

The following is a C code that was generated from the above Mathematica code.

```c
#include <math.h>
#include <stdio.h>

/* Function prototypes */
double a(double, double);
double c(double, double);
double d(double, double);
double f(double, double);
double g(double, double);
double t(double, double);
double b(double, double);
double l(double, double);
double z(double, double);
double alpha0(double, double);
double alpha1(double, double);
double alpha2(double, double);
double alpha3(double, double);
double alpha4(double, double);

/* Global variables */
int hdim, kdim, maxIter, a_loop, u_row, u_col, b_loop, i, m, j;
double epsilon, xMin, xMax, yMin, yMax, h, k, x, y, x_maxE, lastU, difU;
double **u; /* Matrix */

double a (double x, double y)
{
    return (-1);
}

double c (double x, double y)
{
    return (-1);
}

double d (double x, double y)
{
    return (0);
}

double e (double x, double y)
{
    return (0);
}
```
double f (double x, double y) 
{ 
    return (0 );
}

double g (double x, double y) 
{ 
    return ((sin(x) * cos(y)));
}

double t (double x, double y) 
{ 
    return (1 );
}

double b (double x, double y) 
{ 
    return (0 );
}

double l (double x, double y) 
{ 
    return (0 );
}

double r (double x, double y) 
{ 
    return (0 );
}

double alpha0 (double x, double y) 
{ 
    return ( (f(x, y) + (-1 * 2 * a(x, y) / pow(h,2)) + (-1 * 2 * c(x, y) / pow(k,2)) ));
}

double alpha1 (double x, double y) 
{ 
    return ( (a(x, y) / pow(h,2)) + (d(x, y) / 2 * h) );
}

double alpha2 (double x, double y) 
{ 
    return ( (c(x, y) / pow(k,2)) + (e(x, y) / 2 * k) );
}

double alpha3 (double x, double y) 
{ 
    return ( (a(x, y) / pow(h,2)) + (-1 * d(x, y) / 2 * h) );
}

double alpha4 (double x, double y) 
{ 
    return ( (c(x, y) / pow(k,2)) + (-1 * e(x, y) / 2 * k) );
}

int main() 
{
    hdim = 11;
    kdim = 11;
    maxIter = 300;
    epsilon = 1e-06;
}
xMin = 0;
xMax = 1;
yMin = 0;
yMax = 1;

/* Prepare a matrix */
u = (double **) calloc(kdim, sizeof( double *));
for (a_loop=0; a_loop < kdim; a_loop++)
    u[a_loop] = (double *) calloc(hdim, sizeof( double ));

u_row = kdim;
u_col = hdim;
for (a_loop=0; a_loop < u_row; a_loop++)
    for (b_loop=0; b_loop < u_col; b_loop++)
        u[a_loop][b_loop] = 0;

/* Automatically assign zero for initial value */
h = ((xMax + (-1 * xMin)) / (hdim + (-1)));
x = ((yMax + (-1 * yMin)) / (kdim + (-1)));

/* Boundary Values : Only Dirichlet */
for(i=1; i <= hdim; i++)
    { y = (((i + (-1)) * h) + xMin);
      u[0][i-1] = b(xMin, y);
    }
for(i=1; i <= kdim; i++)
    { x = (((i + (-1)) * h) + xMin);
      u[i-1][0] = b(x, yMin);
    }
for(i=1; i <= hdim; i++)
    { y = (((i + (-1)) * h) + yMin);
      u[kdim-1][i-1] = t(xMax, y);
    }
for(i=1; i <= kdim; i++)
    { x = (((i + (-1)) * h) + xMin);
      u[i-1][hdim-1] = t(x, yMax);
    }

for(m=1; m <= maxIter; m++)
    { maxE = 0;
      x = y = 0;
      for(i=2; i <= (kdim + (-1)); i+=1)
        { for(j=2; j <= (hdim + (-1)); j+=1)
            { x = (((i + (-1)) * h) + xMin);
              y = (((j + (-1)) * h) + yMin);
              lastU = u[i-1][j-1];
              u[i-1][j-1] = (g(x, y) +
                        (-1 * alpha1(x, y) * u[(i + (-1))[j-1]] +
                        (-1 * alpha2(x, y) * u[i-1][j+(-1)-1]) +
                        (-1 * alpha3(x, y) * u[(i + (-1)-1)][j-1]] +
                        (-1 * alpha4(x, y) * u[i-1][(j + (-1)-1)]) /
                        alpha0(x, y));
              difU = fabs(lastU + (-1 * u[i-1][j-1]));
              if (maxE < difU)
                { maxE = difU;
                }
            } }
5.2.5 The Result of Execution

The execution of both code showed the same result. They both iterated 110 times until the error (maximum difference in solutions for each iteration) was less than 0.000001. The final solution to the PDE problem can be seen in Figure 8.

5.3 Performance

Since one of the major reasons to generate C code from Mathematica code is to take advantage of the better performance of a compiled language, it is very important to measure the performance of the automatically generated C code. The following figures come from the experiments done on a Sun Sparcstation running SunOS 4.1.3.

The Newton method in Example 1 converged in four steps, and the result was 5.419261. The Mathematica code of Example 1 ran in 0.48 seconds, while the generated C code of Example 1 ran in 0.078 seconds to get the same result.

In Example 2, the C code showed remarkably better performance. It took over 110 seconds to execute the Mathematica code while it took less than 0.48 seconds to execute the compiled C code. The Mathematica code reveals that the dramatic performance increase is due to the fact that there are loops in Mathematica code as described in Section 1.

6 Conclusion

Mathematica is a wonderful computational (either numeric or symbolic) tool engineers and scientists prefer to use for the early development stages of projects. It is not feasible to implement all of their algorithms and execute them in Mathematica since they need the better performance of a compiled language and also need to port Mathematica codes to bigger computational environments. To this end, they have been trying to code the same program in C or Fortran, which is obviously a redundant job.

To lessen the burden of coding twice, McCogen has been developed. McCogen takes entire Mathematica code and generates an entire C program that is free of syntax errors. McCogen also can prevent bugs when programmers port Mathematica code to C.

McCogen has two distinguished features that other code generators do not have.

- McCogen can utilize preexisting function libraries.
- McCogen can generate a whole syntax-free C programs that compiles with no or minimal modifications.

It also has been shown in Section 5 that the execution speed of the generated C code is much faster than the original Mathematica code, especially when there are loops in the Mathematica code.
6.1 Future Work

McCogen should be used for an educational purpose. Students who are exposed to Mathematica programming first like those students in CS158A at Purdue University may learn C programming faster with the help of McCogen.

In addition to that, McCogen should use its own parser and needs to evaluate Mathematica code while translating in order to overcome all shortcomings of the current implementation of McCogen, which have been described in Section 3. That way, it is possible to keep the original precisions of all numbers, to avoid overflow, to estimate data type better, and, as a whole, to translate to better (tight) C code.

To be more specific, McCogen has to be a stand-alone application as a front-end of the Mathematica kernel that takes Mathematica code, parses into appropriate data structures (obviously not FullForm) to keep numbers' original precisions, evaluates necessary the Mathematica code through the Mathematica kernel, and generates better C code.

Lastly, McCogen should use third party libraries for arbitrary precision arithmetic and arithmetic operations on matrices. By providing these library calls as default, the modification to the generated C code will be kept minimal.

7 McCogen Manual

This appendix is a user's manual that describes how to install and use McCogen. Additionally listed are the Mathematica functions currently supported. Functions not on the list cannot be used to generate C code except resolvable symbolic functions like D, Sum, Integrate, etc.

7.1 Installation

1. First get the distribution file.

   \texttt{ftp.cs.purdue.edu:/pub/song/Mccogen.tar.gz}

2. Unpack the distribution file.

   \texttt{gunzip -c McCogen.tar.gz | tar xvf -}

3. \texttt{cd McCogen}, and make necessary modifications to Makefile.

4. In Makefile, at least first two variables, MATHPATH and CC, have to be set correctly.

   (a) \texttt{MATHPATH} should point to the root directory of Mathematica distribution.
       Default: /usr/local/math

   (b) \texttt{CC} should point to a ANSI C compiler. Default: gcc

5. Type make.

6. Now an executable named \texttt{m2c} should be in the current directory.

7.2 Running

Now you are ready to run McCogen.

7.2.1 Install

In Mathematica, you can type

\begin{verbatim}
link=Install["m2c"]
\end{verbatim}

if the executable, m2c, is in the current directory. Otherwise prepend the full path name.
7.2.2 Running from other hosts

If you want to run McCogen from other hosts, follow the procedure. Suppose McCogen will be running on express.cs.purdue.edu and a Mathematica kernel is running on icarus.cs.purdue.edu.

1. Manually run McCogen by typing m2c on shell prompt on express. Then type a TCP port number McCogen should listen to. Suppose the user typed 3000.

2. On icarus.cs.purdue.edu, issue the following command in Mathematica.

   ```mathematica
   link = Install["3000@express.cs.purdue.edu", LinkMode -> Connect]
   ```

7.2.3 Disconnect

The command Uninstall will disconnect McCogen with the Mathematica kernel. This command will gracefully shut down McCogen process.

   ```mathematica
   Uninstall[link]
   ```

7.2.4 How to write to a file?

Use a mouse to copy the output cell that contains the generated C code and paste them into an editor. Alternatively you can use the following command to write to a file directly.

   ```mathematica
   OutputForm[
   M2C
   ] >> file.c
   ```

7.3 A List of currently supported Mathematica functions and descriptions

This list will always be changing as long as McCogen is in alpha or beta version.

- Boolean constants

   ```plaintext
   TRUE FALSE
   ```

- These functions will be ignored unless they are inside symbolic functions.

   ```plaintext
   Hold Clear N[] //N
   ```

- Assignment statements (= or Set). The following form of assignments works.
But the following form doesn't work.

(a, b, c) = (1, 3, 2)

- Most arithmetic operators

+ - * / Mod  ++  --  +=  -=  /=  +=

- Some trigonometric functions

Sin  Cos  Tan  ArcTan  ArcCos  Sinh  Cosh  Tanh

- Some basic arithmetic functions

Power  Log  Abs

- Some constants

PI  E

- Comment

Comment()

will put comments into C code.

- Print will be translated to printf with correct conversion specifications.

7.4 Scalar Data Types

Integer and real numbers are limited to int and double. Since in the current version of MathLink, real numbers are treated as double in C, all real numbers will be declared as double. So if users type very big numbers, it is most likely that the MathLink connection will brake. If it does, kill the m2c process manually. Complex numbers are not supported.
7.5 String

To assign a string to a variable, in Mathematica, simply using := operator does the job. But in C, simply using assignment statement won't work. Assignment of a string to a variable will be translated into a couple of C statements including a memory allocation and a string copy. For example,

```
In[11]=
M2C[ b="Hello"; Print[b] ]

Out[11]=

#include <string.h>
#include <stdio.h>

/* Global variables */
char *b;

int main()
{
    b = (char *) calloc(strlen("Hello")+1, sizeof(char));
    strcpy(b, "Hello");
    printf("%s\n", b);
}
```

Assignment from a variable that was assigned a string also handled by = operator in Mathematica. The situation also is treated in the same way in C. For example,

```
In[1]=
M2C[a="Hello, World!"; b=a; c=b]

Out[1]=

#include <string.h>

/* Global variables */
char *a, *b, *c;

int main()
{
    a = (char *) calloc(strlen("Hello, World!")+1, sizeof(char));
    strcpy(a, "Hello, World!");
    b = (char *) calloc(strlen(a), sizeof(char)); strcpy(b, a);
    c = (char *) calloc(strlen(b), sizeof(char)); strcpy(c, b);
}
```

7.6 Vectors and matrices

McCogen supports dynamic memory allocation for vectors and matrices. Their initializations can be done using either lists directly or Table function. Vectors will be declared as pointer variables, and matrices will be declared as pointers to pointer variables. The size of a vector will be stored in a variable whose name is the name of the vector with _size appended. The row and column size of a matrix will be stored in corresponding variables with _row and _col appended to the original variable name. So vector_size will substitute for Length[vector] in generated C code. For matrix, currently there is no way of getting the size of rows and columns of a matrix.

The feature of getting row and column with Part[Dimensions[matrix], i] will be added later.
Vectors
Matrix

In(1):=
M2C[a=((1,2), (3,4,5), (4,5,6)); b=(12.34, 5.543, 5.0, 5.45);
i=j=0; a[[1,2]] = 4.3;
Do[total = a[[i,j]] + a[[j,i]], {i, j}]
]

Out(1):=
/* Global variables */
int a_row, a_col, b_size, i, j, for_count;
double total;
double *b; /* Vectors */
double **a; /* Matrix */

int main()
{
a = (int **) calloc(3, sizeof( int *));
a[0] = (int *) calloc(3, sizeof( int ));
a[1] = (int *) calloc(3, sizeof( int ));
a[2] = (int *) calloc(3, sizeof( int ));
a_row = 3;
a_col = 3;
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 3;
a[1][1] = 4;
a[1][2] = 5;
a[2][0] = 4;
a[2][1] = 5;
a[2][2] = 6;
b = (double *) calloc(4, sizeof( double ));
b_size = 4;
b[0] = 12.34;
b[1] = 5.543;
b[2] = 5;
b[3] = 5;
i = j = 0;
for(for_count=0; for_count <3; for_count++)
{
    total = (a[i-1][j-1] + a[j-1][i-1]);
}
}

In(2):=
M2C[i=1; j=1;
a=Table[ 1/(i++ + j++), {10},{20}];
b=Table[0, {20}];a[[i-1,j-2]]=3]

Out(2):=
/* Global variables */
int i, j, a_row, a_col, b_loop, b_size;
int *b; /* Vectors */
double **a; /* Matrix */

int main()
{
i = 1;
j = 1;
a = (double **) calloc(10, sizeof( double *));
for (a_loop=0; a_loop < 10; a_loop++)
a[a_loop] = (double *) calloc(20, sizeof( double ));

7.7 Boolean Operators

All boolean operators are supported, and they are

\[
\begin{align*}
&= \quad \neq \quad < \quad > \quad \leq \quad \geq \\
\land \quad \lor \quad !
\end{align*}
\]

In Mathematica, one can do

\[
\text{If}[1<i<6, a=1]
\]

In other words, more than one boolean operators can be applied in a cascade fashion. It is not possible in C, but similar C code will be placed to simulate the Mathematica code. For example,

\[
\text{In}[6]:=
\text{M2C}(i=1; j=2; k=3; \text{If}[1<i<2 \land i!=j \land i!=k \land j!=k, \text{Not}[k], \text{Print}[i]])
\]

\[
\text{Out}[6]=
\text{include <stdio.h>}

/* Global variables */

int i, j, k;

int main()
{
    i = 1;
    j = 2;
    k = 3;
    if (((1 < i && i < 2) && (i != j && i != k && j != k)) || (!k))
    {
        printf("%d\n", i);
    }
}

7.8 Control structures

All control structure related functions are supported. They are
Nested loops are supported, too. For DO loops, if a loop counter variable is not specified, for_counter will be used as a loop counter, and \_ will be added if the same name is already in use. For example,

```c
#include <stdio.h>

/* Global variables */
int total, for_count, for_count_

int main()
{
    total = 0;
    for (for_count=0; for_count < 10; for_count++)
    {
        total++;
        for (for_count_=0; for_count_ < 20; for_count_++)
        {
            total--;
        }
    }
    printf("%d\n", total);
}
```

### 7.9 Function declaration

Mathematica functions can be defined using `:=` operator or `SetDelayed`, which is the same thing. To use local variables `Module` can be used. Block and With are not supported. With support will be added later.

Since functions can be defined in various ways, only the following forms are supported.

```mathematica
f[x_Real] := x^2
f[x_Real] := (a=1; b=a)
```

Several definitions for one function cannot be used.

```mathematica
f[0] := 3; f[x_Integer] := x^2
```

Conditional definition is not supported, either.

```mathematica
f[x_Integer] := x^2 /; x>2
```
7.10 Using Own Libraries

It is possible to use preexisting libraries. Users have to include the function calls in Mathematica code as if they were built-in Mathematica functions. The following command forces McCogen to recognize the function as built-in function.

```
M2C[ExistingFunction["func_name", "header_file_name", 
(input_parameter_types), (return_type)]
```

For example,

```
M2C[ExistingFunction["sum", "my_math.h", 
(Integer, Integer, Double), (Double)]
```

In the above example, the user wants to use a C function sum() that accepts two integers and one double number and returns a double number, and its prototype is in "my_math.h".

Then until Uninstall is issued, the function named func.name can be used in generated C code.

```
In[13]:= link=Install["m2c"]
Out[13]= LinkObject[m2c, 12, 1]

In[14]:= M2C[ExistingFunction["harmonic", "mathfunc.h", {Integer}, (Double)]]
Out[14]=

```
Global variables

```
int main()
{
    ...
}
```

```
In[15]:= M2C[a=x=25; b=harmonic[Sin[a]+Integrate[x^2,x]]; Print[b]]
Out[15]=
#include <math.h>
#include "mathfunc.h"
#include <stdio.h>

/* Global variables */
int a, x;
double b;

int main()
{
    a = x = 25;
    b = harmonic((int) (sin(a) + Integrate(x^2,x))); Print(b);
    printf("%f\n", b);
}
```

7.11 Symbolic functions

The evaluation of symbolic functions is done in a greedy fashion: when McCogen identifies unknown functions, it treats the functions as symbolic functions and forces the kernel to evaluate the functions. And McCogen tries to translate the
resulting expressions into C code. It does the greedy evaluation recursively to resolve all unknown functions until the kernel's iteration limit is reached. At this point it is better to install McCogen again.

Users can use symbolic functions to define functions. Symbolic functions can also be applied to user-defined functions. For example,

```c
#include <math.h>

/* Global variables */
int x, y;
double a, e, b, c, d;

int main()
{
    x = 1;
    y = 2;
    a = ((0.5 * pow(x,2)) + (0.3333333333 * pow(x,3)) + (x * y));
    e = 3.8333333333;
    b = ((cos(log(cos(x))) * tan(x)) + 2.358409254);
    c = (1 + x + (0.5 * pow(x,2)) + (0.1666666667 * pow(x,3)) +
         (0.0416666667 * pow(x,4)) + (0.008333333333 * pow(x,5)));
    d = 1;
}
```

```c
In[10]:=
M2C[x=1;y=2;
   a = Integrate[x^2+y,x]; e = NIntegrate[x^2,x,{x,0,10}];
   b = D[Sin[Integrate[Tan[x],x]],x] + Sum[1/x,{x,10,100}];
   c = Normal[Series[Exp[x],{x,0,5}]];
   d = Limit[ Sin[x]/x, x->0]
]

Out[10]:=

#include <math.h>

/* Global variables */
int x, y;
double a, e, b, c, d;

int main()
{
    x = 1;
    y = 2;
    a = ((0.5 * pow(x,2)) + (0.3333333333 * pow(x,3)) + (x * y));
    e = 3.8333333333;
    b = ((cos(log(cos(x))) * tan(x)) + 2.358409254);
    c = (1 + x + (0.5 * pow(x,2)) + (0.1666666667 * pow(x,3)) +
         (0.0416666667 * pow(x,4)) + (0.008333333333 * pow(x,5)));
    d = 1;
}
```

```c
In[11]:=
M2C[ f[x_Double] := 4000 Pi / x + 1; r=0; y=3; x=D[f[r],r]]

Out[11]:=

#include <math.h>

/* Function prototypes */
double f ( double );

/* Global variables */
int r, y;
double x;

double f ( double x )
{
    return (((4000 * M_PI / x) + 1) );
}

int main()
{
    z = 0;
    y = 3;
    x = (-4000 * M_PI - pow(r,-2));
}
References


Figure 8: Plots of the solution to the PDE problem in Example 2